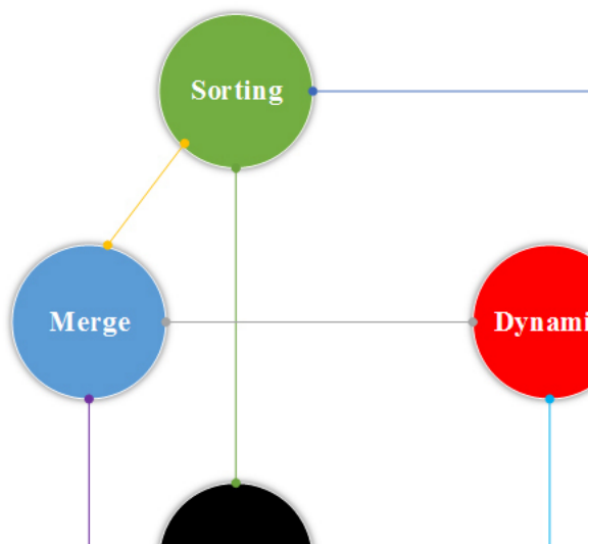


Algorit. C#



Algorithms C#



YANG HU

Simple is the beginning of wisdom. From the essence of practice, this book to briefly explain the concept and vividly cultivate programming interest, you will learn it easy and fast.

<http://en.verejava.com>

Copyright © 2020 Yang Hu

All rights reserved.

ISBN : 979867125286 6

CONTENTS

- 1 . [Linear Table Definition](#)
- 2 . [Maximum Value](#)
3. [Bubble Sorting Algorithm](#)
- 4 . [Minimum Value](#)
5. [Select Sorting Algorithm](#)
6. [Linear Table Append](#)
7. [Linear Table Delete](#)
8. [Insert Sorting Algorithm](#)

- 9 . Reverse Array
10. Linear Table Search
11. Dichotomy Binary Search
- 12 . Shell Sorting
13. Unidirectional Linked List
 - 13.1 Create and Initialization
 - 13.2 Add Node
 - 13.3 Insert Node
 - 13.4 Delete Node
14. Doubly Linked List
 - 14.1 Create and Initialization
 - 14.2 Add Node
 - 14.3 Insert Node
 - 14.4 Delete Node
15. One-way Circular LinkedList
 - 15.1 Initialization and Traversal
 - 15.2 Insert Node
 - 15.3 Delete Node
16. Two-way Circular LinkedList
 - 16.1 Initialization and Traversal
 - 16.2 Insert Node
 - 16.3 Delete Node
17. Queue
18. Stack
19. Recursive Algorithm
20. Two-way Merge Algorithm
21. Quick Sort Algorithm
22. Binary Search Tree
 - 22.1 Construct a binary search tree
 - 22.2 Binary search tree In-order traversal
 - 22.3 Binary search tree Pre-order traversal

- 22.4 Binary search tree Post-order traversal
- 22.5 Binary search tree Maximum and minimum
- 22.6 Binary search tree Delete Node
- 23. Binary Heap Sorting
- 24. Hash Table
- 25. Graph
 - 25.1 Directed Graph and Depth-First Search
 - 25.2 Directed Graph and Breadth-First Search
 - 25.3 Directed Graph Topological Sorting
- 26 . Towers of Hanoi
- 27 . Fibonacci
- 28 . Dijkstra
- 29 . Mouse Walking Maze
- 30 . Eight Coins
- 31 . Knapsack Problem
- 32 . Josephus Problem

Linear Table Definition

Linear Table:

Sequence of elements, is a one-dimensional array.

1 . Define a one-dimensional array of student scores

0	1	2	3	4	5
90	70	50	80	60	85

length = 6

TestOneArray.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestOneArray
    {
        static void Main(string [] args)
        {
            int [] scores = { 90, 70, 50, 80, 60, 85 };
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.WriteLine(scores[i] + ",");
            }
        }
    }
}
```

Result:

90,70,50,80,60,85,

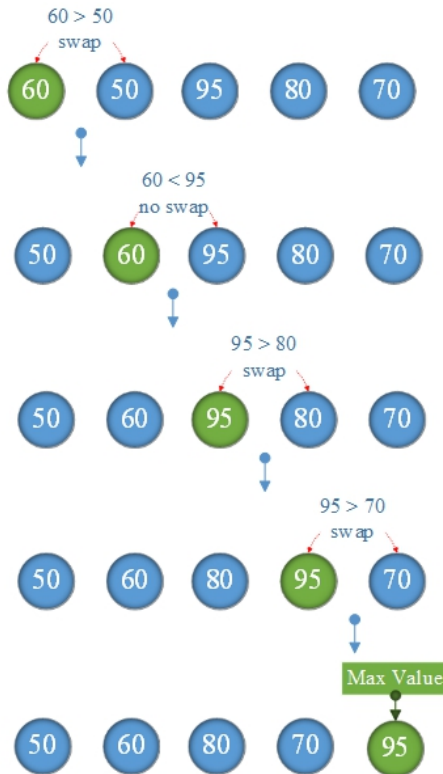
Maximum Value

Maximum of Integer Sequences:



1 . Algorithmic ideas

Compare $\text{arrays}[i]$ with $\text{arrays}[i + 1]$, if $\text{arrays}[i] > \text{arrays}[i + 1]$ are exchanged. So continue until the last number, $\text{arrays}[\text{length} - 1]$ is the maximum.



TestMaxValue.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestMaxValue
    {
        public static void Main(String [] args)
        {
            int [] scores = { 60, 50, 95, 80, 70 };
            int maxVal = Max(scores);
            Debug.WriteLine("Max Value = " + maxVal);
        }
        public static int Max(int [] arrays)
        {
            // Maximum initialization value is 0
            for (int i = 0; i < arrays.Length - 1; i++)
            {
                if (arrays[i] > arrays[i + 1])
                {
                    int temp = arrays[i];
                    arrays[i] = arrays[i + 1];
                    arrays[i + 1] = temp;
                }
            }
            int maxVal = arrays[arrays.Length - 1];
            return maxVal;
        }
    }
}
```

Result:

Max Value = 95

Bubble Sorting Algorithm

Bubble Sorting Algorithm:

Compare `arrays[j]` with `arrays[j + 1]` , if `arrays[j] > arrays[j + 1]` are exchanged.

Remaining elements repeat this process, until sorting is completed.

Sort the following numbers from small to large



Explanation:



No sorting,

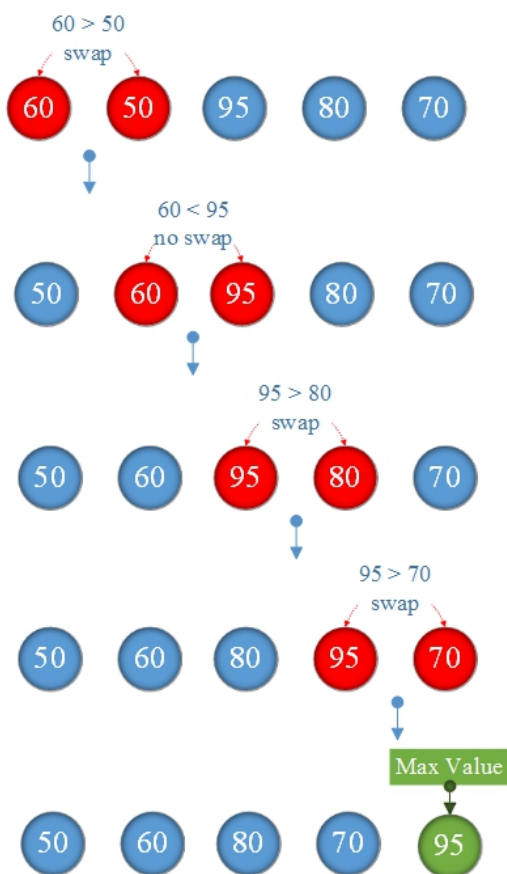


Comparing,

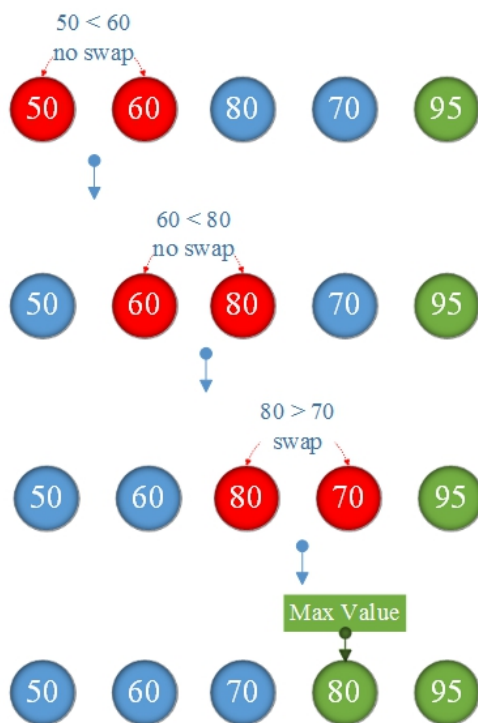


Already sorted

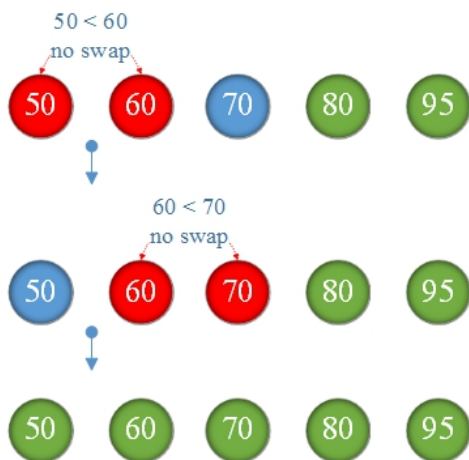
1 . First sorting:



2 . Second sorting:



3 . Third sorting:



No swap so terminate sorting : we can get the sorting numbers from small to large



TestBubbleSort.cs

```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class TestBubbleSort
    {
        static void Main(string [] args)
        {
            int [] scores = { 90, 70, 50, 80, 60, 85 };
            Sort(scores);
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.WriteLine(scores[i] + ",");
            }
        }
        public static void Sort(int [] arrays)
        {
            for (int i = 0; i < arrays.Length - 1; i++)
            {
                for (int j = 0; j < arrays.Length - i - 1; j++)
                {
                    //swap
                    if (arrays[j] > arrays[j + 1])
                    {
                        int flag = arrays[j];
                        arrays[j] = arrays[j + 1];
                        arrays[j + 1] = flag;
                    }
                }
            }
        }
    }
}
```

Result:

50,60,70,80,85,90,

Minimum Value

Search the Minimum of Integer Sequences:

60

80

95

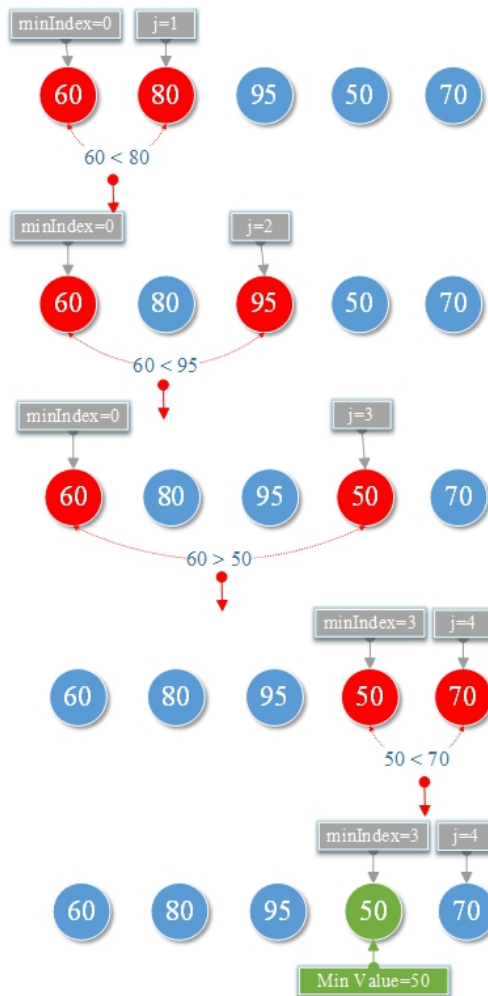
50

70

1 . Algorithmic ideas

Initial value `minIndex = 0, j = 1` Compare `arrays[minIndex]` with `arrays[j]`

if `arrays[minIndex] > arrays[j]` then `minIndex = j, j++` else `j++`.
continue until the last number, `arrays[minIndex]` is the Min Value.



TestMinValue.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestMinValue
    {
        public static void Main(String [] args)
        {
            int [] scores = { 60, 80, 95, 50, 70 };
            int minValue = Min(scores);
            Debug.WriteLine("Min Value = " + minValue);
        }
        public static int Min(int [] arrays)
        {
            int minIndex = 0; // the index of the minimum
            for (int j = 1; j < arrays.Length; j++)
            {
                if (arrays[minIndex] > arrays[j])
                {
                    minIndex = j;
                }
            }
            return arrays[minIndex];
        }
    }
}
```

Result:

Min Value = 50

Select Sorting Algorithm

Select Sorting Algorithm:

Sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning.

Sort the following numbers from small to large



Explanation:



No sorting,

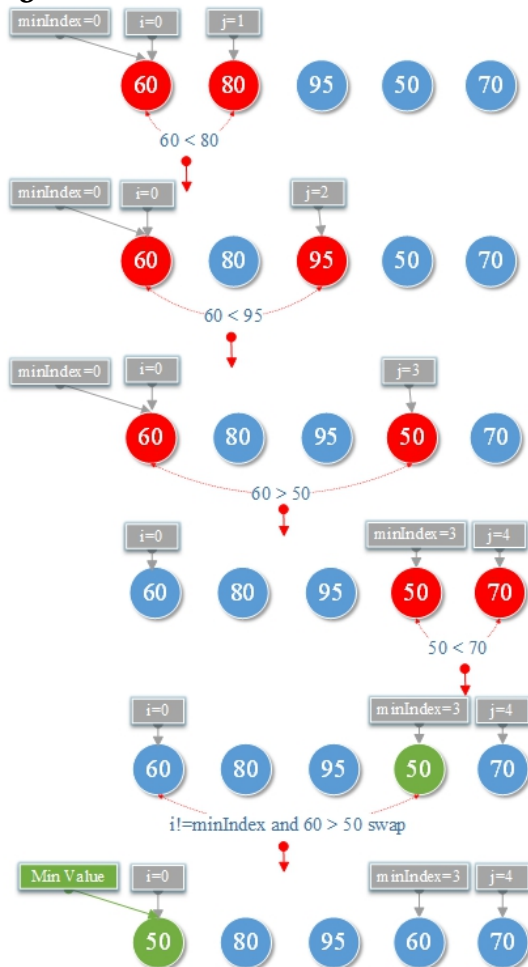


Comparing,

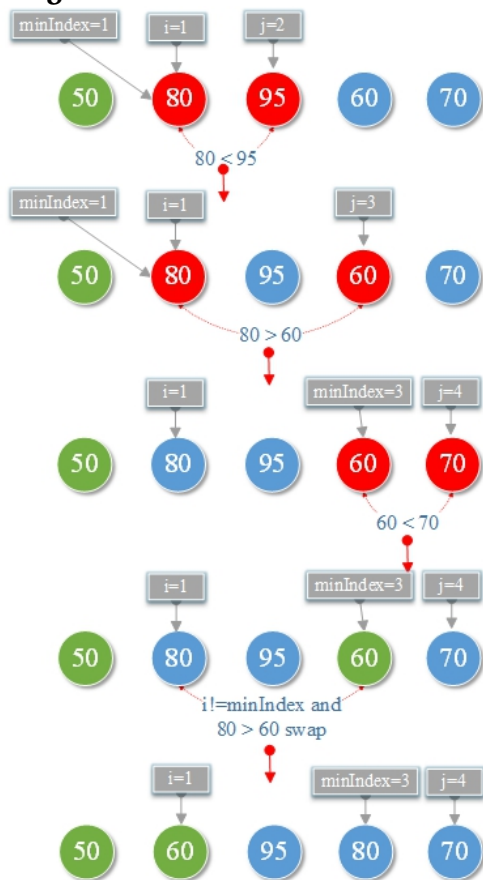


Already sorted.

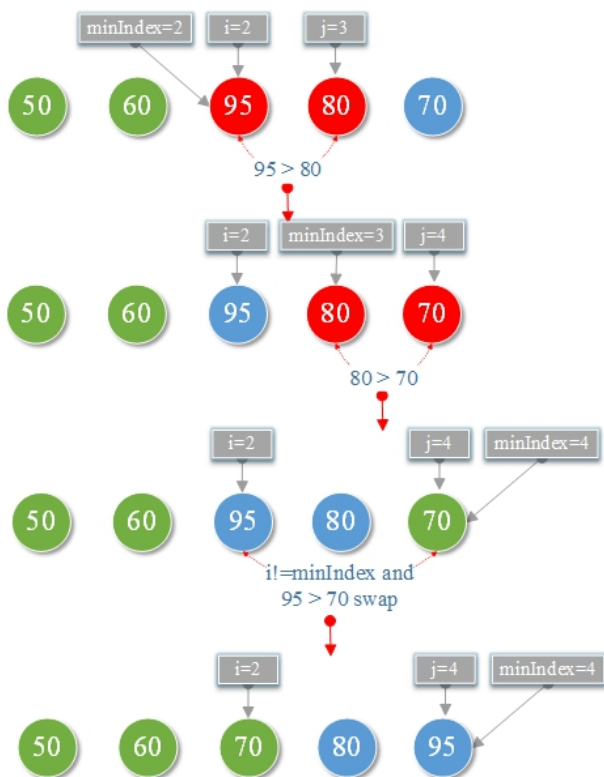
1 . First sorting:



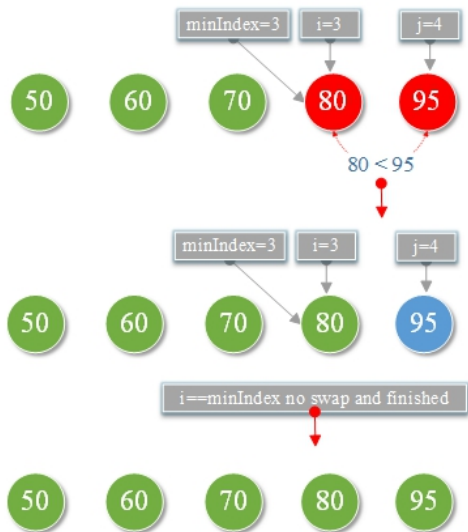
2 . Second sorting:



3 . Third sorting:



4 . Forth sorting:



we can get the sorting numbers from small to large

50 60 70 80 95

TestSelectSort.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestSelectSort
    {
        public static void Main(string [] args) {
            int [] scores = { 90, 70, 50, 80, 60, 85 };
            Sort(scores);
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.WriteLine(scores[i] + ",");
            }
        }
        public static void Sort(int [] arrays)
        {
            int len = arrays.Length - 1;
            int minIndex; // Save the index of the selected minimum
            for (int i = 0; i < len; i++)
            {
                minIndex = i;
                int minValue = arrays[minIndex];
                for (int j = i; j < len; j++)
                {
                    if (minValue > arrays[j + 1]) // minimum
                    exchange with minIndex
                    {
                        minValue = arrays[j + 1];
                        minIndex = j + 1;
                    }
                }
                if (i != minIndex)
                {
                    int temp = arrays[i];
                    arrays[i] = arrays[minIndex];
                    arrays[minIndex] = temp;
                }
            }
        }
    }
}
```

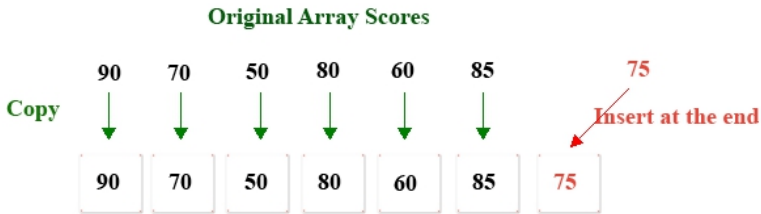
}
}
}
}
}

Result:

50,60,70,80,85,90,

Linear Table Append

1 . Add a score **75** to the end of the one-dimensional array scores.



Analysis:

1. First create a temporary array(**tempArray**) larger than the original scores array length
2. Copy each value of the scores to **tempArray**
3. Assign 75 to the last index position of **tempArray**
4. Finally assign the **tempArray** pointer reference to the original scores;

TestOneArrayAppend.cs

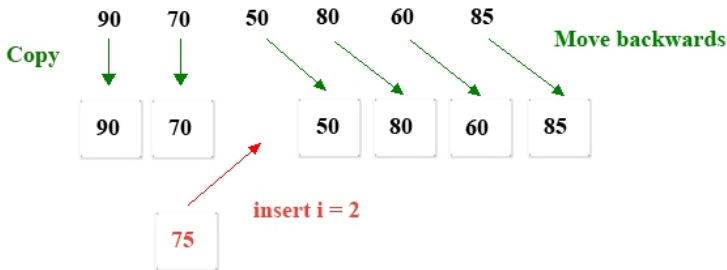
```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class TestOneArrayAppend
    {
        static int [] Append(int [] array, int value)
        {
            //create a new array, Length = array.Length + 1
            int [] tempArray = new int [array.Length + 1];
            for (int i = 0; i < array.Length; i++)
            {
                tempArray[i] = array[i];
            }
            tempArray[array.Length] = value;
            return tempArray;
        }
        static void Main(string [] args)
        {
            int [] scores = { 90, 70, 50, 80, 60, 85 };
            scores = Append(scores, 75);
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.WriteLine(scores[i] + ",");
            }
        }
    }
}
```

Result:

90,70,50,80,60,85,75,

Linear Table Insert

1 . Insert a student's score anywhere in the one-dimensional array scores.



Analysis:

1. First create a temporary array **tempArray** larger than the original scores array length
2. Copy each value of the previous value of the scores array from the beginning to the insertion position to **tempArray**
3. Move the scores array from the insertion position to each value of the last element and move it back to **tempArray**
4. Then insert the score 75 to the index of the **tempArray** .
5. Finally assign the **tempArray** pointer reference to the scores;

TestOneArrayInsert.cs

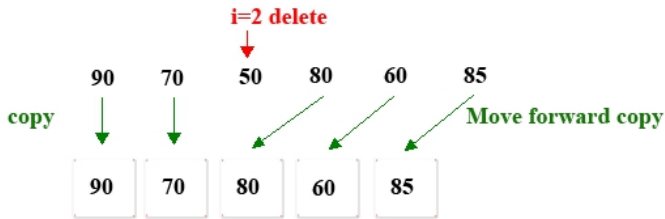
```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class TestOneArrayInsert
    {
        public static int [] Insert(int [] array, int score, int
insertIndex)
        {
            int [] tempArray = new int [array.Length + 1];
            for (int i = 0; i < array.Length; i++)
            {
                if (i < insertIndex)
                {
                    tempArray[i] = array[i];
                }
                else
                {
                    tempArray[i + 1] = array[i];
                }
            }
            tempArray[insertIndex] = score;
            return tempArray;
        }
        public static void Main(string [] args)
        {
            int [] scores = { 90, 70, 50, 80, 60, 85 };
            scores = Insert(scores, 75, 2); //Insert 75 into the
position: index = 2
            for (int i = 0; i < scores.Length; i++)
            {
                Debug .Write(scores[i] + ",");
            }
        }
    }
}
```

Result:

90,70,75,50,80,60,85,

Linear Table Delete

1 . Delete the value of the **index = 2** from scores array



Analysis:

1. Create a temporary array **tempArray** that length smaller than scores by 1.
2. Copy the data in front of **i = 2** to the front of **tempArray**
3. Copy the array after **i = 2** to the end of **tempArray**
4. Assign the **tempArray** pointer reference to the scores
5. Printout scores

TestOneArrayDelete.cs

```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class TestOneArrayDelete
    {
        public static int [] Delete(int [] array, int index)
        {
            // create a new array, Length = array.Length - 1
            int [] tempArray = new int [array.Length - 1];
            for (int i = 0; i < array.Length; i++)
            {
                if (i < index) // Copy the data in front of index to the
front of tempArray
                    tempArray[i] = array[i];
                if (i > index) // Copy the array after index to the
end of tempArray
                    tempArray[i - 1] = array[i];
            }
            return tempArray;
        }
        public static void Main(string [] args)
        {
            int [] scores = { 90, 70, 50, 80, 60, 85 };
            Debug .Write("Please enter the index to be deleted:");
            int index = Convert .ToInt32(Console .ReadLine());
            scores = Delete(scores, index);
            for (int i = 0; i < scores.Length; i++)
            {
                Debug .Write(scores[i] + ",");
            }
        }
    }
}
```

Result:

Please enter the index to be deleted:

2

90,70,80,60,85,

Insert Sorting Algorithm

Insert Sorting Algorithm:

Take an unsorted new element in the array, compare it with the already sorted element before, if the element is smaller than the sorted element, insert new element to the right position.

Sort the following numbers from small to large



Explanation:



No sorting,

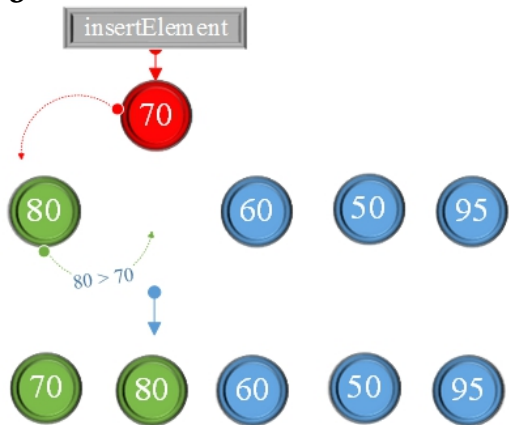


Inserting,

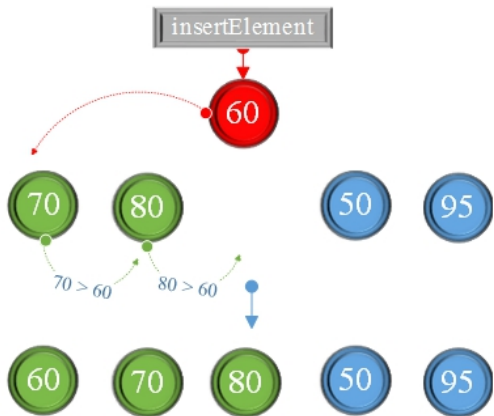


Already sorted

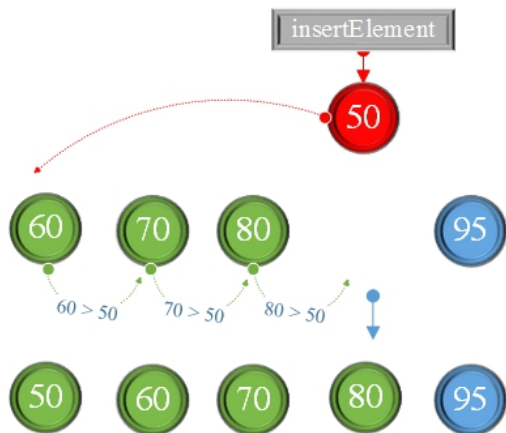
1 . First sorting:



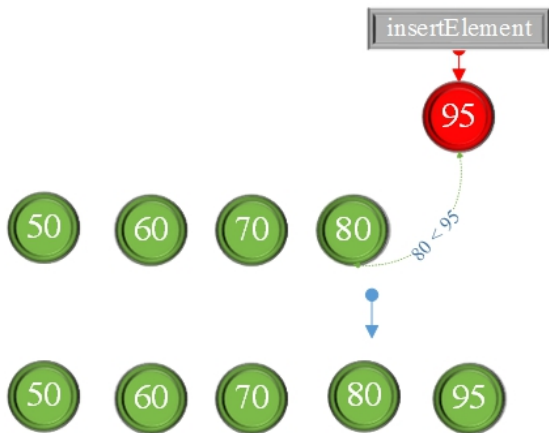
2 . Second sorting:



3 . Third sorting:



4 Third sorting:



TestInsertSort.cs

```
using System;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestInsertSort
    {
        public static void Main(string [] args)
        {
            int [] scores = { 90, 70, 50, 80, 60, 85 };
            sort(scores);
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.WriteLine(scores[i] + ",");
            }
        }
        public static void sort(int [] arrays)
        {
            for (int i = 0; i < arrays.Length; i++) {
                int insertElement = arrays[i]; //Take unsorted new
                elements
                int insertPosition = i; //Inserted position
                for (int j = insertPosition - 1; j >= 0; j--) {
                    //If the new element is smaller than the sorted
                    element, it is shifted to the right
                    if (insertElement < arrays[j]) {
                        arrays[j + 1] = arrays[j];
                        insertPosition--;
                    }
                }
                arrays[insertPosition] = insertElement; //Insert the
                new element
            }
        }
    }
}
```

Result:

50,60,70,80,85,90,

Reverse Array

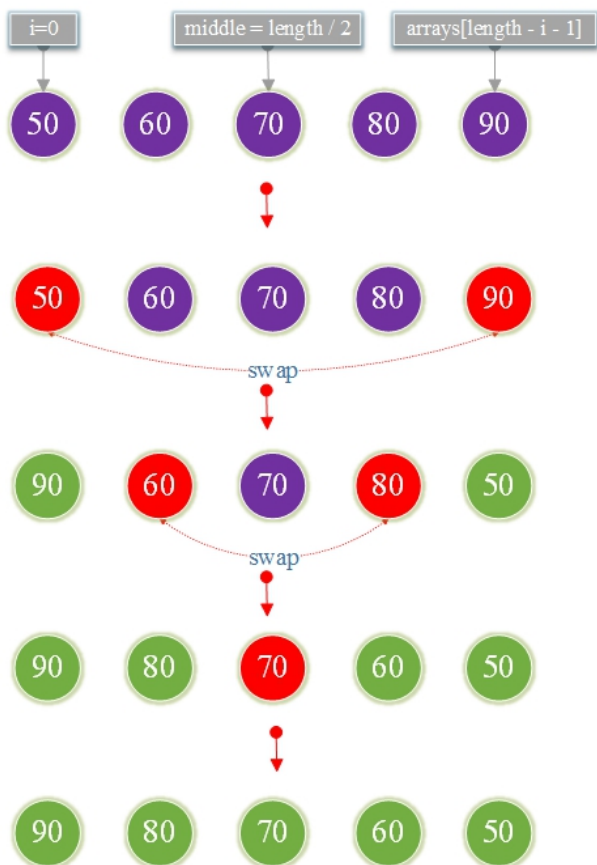
Inversion of ordered sequences:



1 . Algorithmic ideas

Initial $i = 0$ and then swap the first element `arrays[i]` with last element `arrays[length - i - 1]`

Repeat until index of middle $i == \text{length} / 2$.



TestReverse.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestReverse
    {
        public static void Main(String [] args)
        {
            int [] scores = { 50, 60, 70, 80, 90 };
            Reverse(scores);
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.WriteLine(scores[i] + ",");
            }
        }
        public static void Reverse(int [] arrays)
        {
            int length = arrays.Length;
            int middle = length / 2;
            for (int i = 0; i <= middle; i++) {
                int temp = arrays[i];
                arrays[i] = arrays[length - i - 1];
                arrays[length - i - 1] = temp;
            }
        }
    }
}
```

Result:

90,80,70,60,50,

Linear Table Search

1 . Please enter the value you want to search like : **70** return index.



Analysis:

Traverse the value in the array scores, if there is a value equal to the given value like **70** , print out the current index

TestOneArraySearch.cs

```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class TestOneArraySearch
    {
        public static int Search(int [] array, int value)
        {
            for (int i = 0; i < array.Length; i++)
            {
                if (array[i] == value)
                {
                    return i;
                }
            }
            return -1;
        }
        public static void Main(string [] args)
        {
            int [] scores = { 90, 70, 50, 80, 60, 85 };
            Debug.WriteLine("Please enter the value you want to search
:");
            int value = Convert.ToInt32(Console.ReadLine());
            int index = Search(scores, value);
            if (index > 0)
            {
                Debug.WriteLine("Found value: " + value + " the index
is: " + index);
            }
            else
            {
                Debug.WriteLine("The value was not found : " + value);
            }
        }
    }
}
```

Result:

Please enter the value you want to search :

70

Found value: 70 the index is: 1

Dichotomy Binary Search

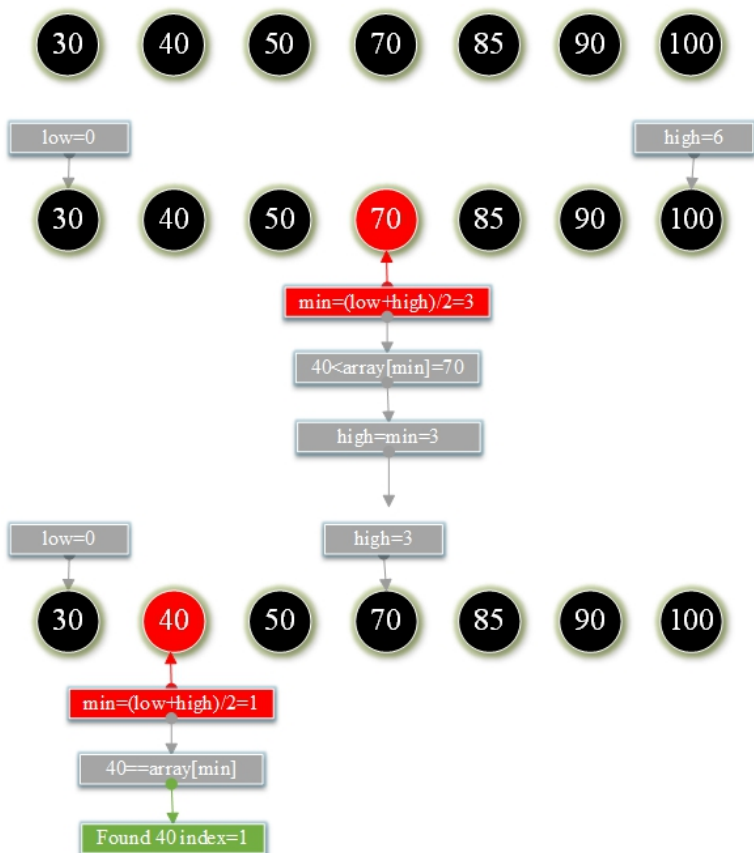
Dichotomy Binary Search:

Find the index position of a given value from an already ordered array.

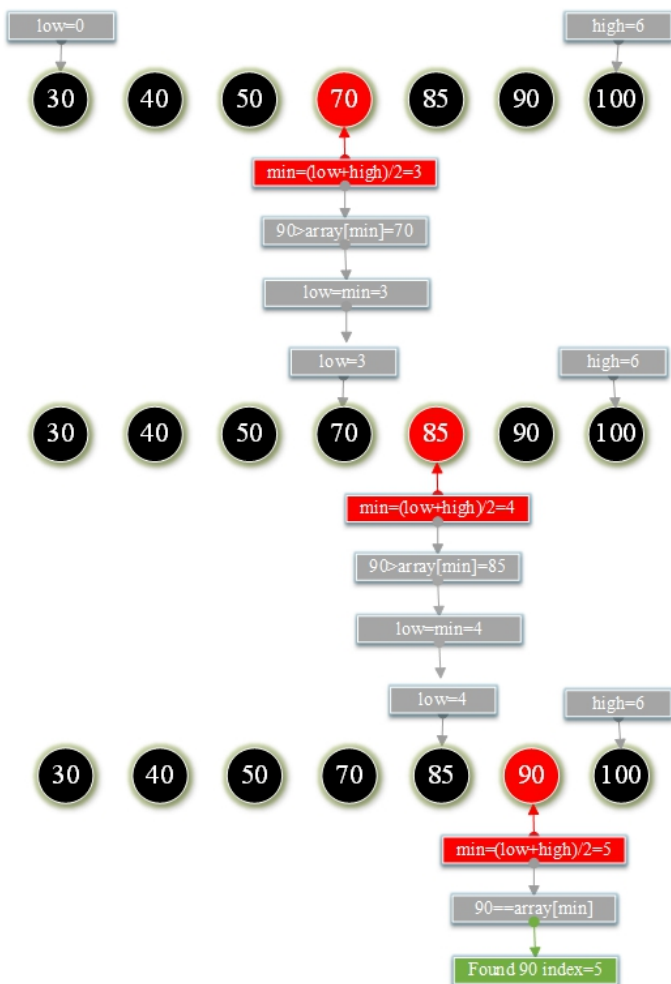


1. Initialize the lowest index **low = 0** , the highest index **high = scores.length-1**
2. Find the **searchValue** of the middle index **mid = (low + high)/2**
scores[mid]
3. Compare the **scores[mid]** with **searchValue**
If the **scores[mid] == searchValue** print current mid index,
If **scores[mid] > searchValue** that the **searchValue** will be found
between **low and mid-1**
4. And so on. Repeat step 3 until you find **searchValue** or
low > = high to terminate the loop.

Example 1 : Find the index of **searchValue = 40** in the array that has been sorted below.



Example 2 : Find the index of **searchValue = 90** in the array that has been sorted below.



TestBinarySearch.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestBinarySearch
    {
        public static void Main(string [] args)
        {
            int [] scores = { 30, 40, 50, 70, 85, 90, 100 };
            int searchValue = 40;
            int position = BinarySearch(scores, searchValue);
            Debug.Write(searchValue + " position:" + position);
            Debug.WriteLine("\n-----\n");
            searchValue = 90;
            position = BinarySearch(scores, searchValue);
            Debug.WriteLine(searchValue + " position:" + position +
"\n");
        }
        public static int BinarySearch(int [] arrays, int searchValue)
        {
            int low = 0;
            int high = arrays.Length - 1;
            int mid = 0;
            while (low <= high)
            {
                mid = (low + high) / 2;
                if (arrays[mid] == searchValue)
                {
                    return mid;
                }
                else if (arrays[mid] < searchValue)
                {
                    low = mid + 1;
                }
                else if (arrays[mid] > searchValue)
                {

```

```
        high = mid - 1;
    }
}
return -1;
}
}
```

Result:

40 position:1

90 position:5

Shell Sorting

Shell Sorting:

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

Sort the following numbers from small to large by Shell Sorting

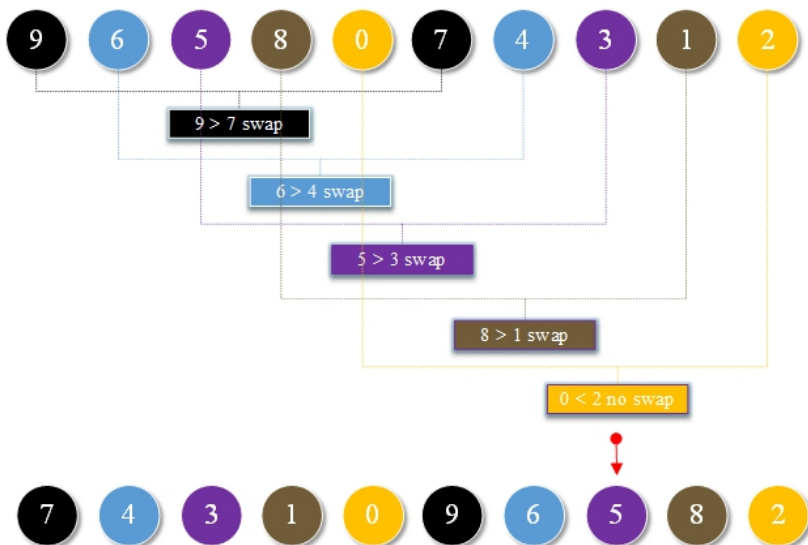


Algorithmic result:

The array is grouped according to a certain increment of subscripts, and the insertion of each group is sorted. As the increment decreases gradually until the increment is 1, the whole data is grouped and sorted.

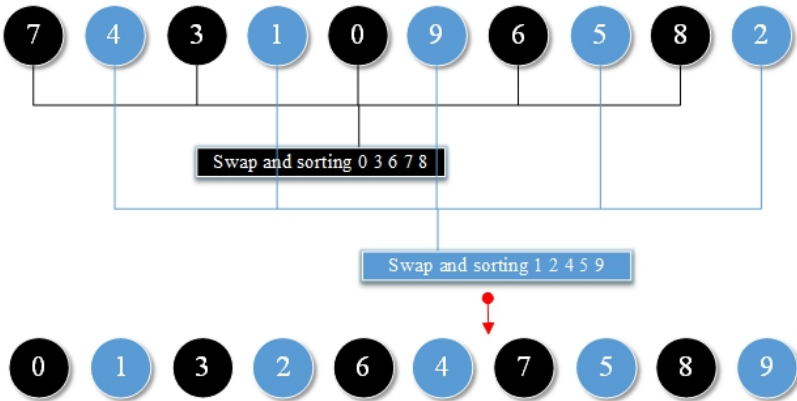
1 . The first sorting :

$\text{gap} = \text{array.length} / 2 = 5$



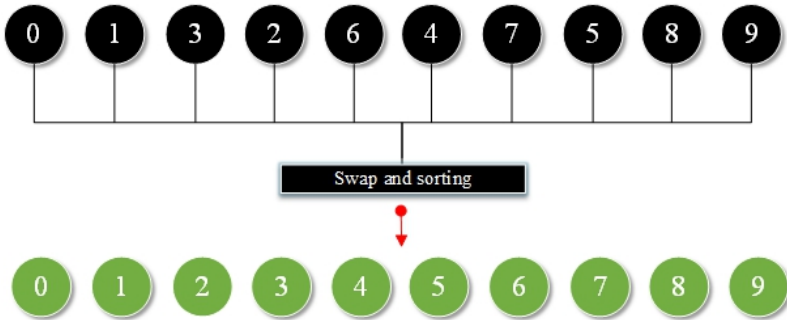
2 . The second sorting :

$\text{gap} = 5 / 2 = 2$



3 . The third sorting :

$$\text{gap} = 2 / 2 = 1$$



TestShellSort.cs

```
using System;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestShellSort
    {
        public static void Main(String [] args)
        {
            int [] scores = { 9, 6, 5, 8, 0, 7, 4, 3, 1, 2 };
            ShellSort(scores);
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.WriteLine(scores[i] + ",");
            }
        }
        public static void ShellSort(int [] array)
        {
            for (int gap = array.Length / 2; gap > 0; gap = gap /
2)
            {
                for (int i = gap; i < array.Length; i++)
                {
                    int j = i;
                    while (j - gap >= 0 && array[j] < array[j - gap])
                    {
                        Swap(array, j, j - gap);
                        j = j - gap;
                    }
                }
            }
        }
        public static void Swap(int [] array, int a, int b)
        {
            array[a] = array[a] + array[b];
            array[b] = array[a] - array[b];
            array[a] = array[a] - array[b];
        }
    }
}
```

}

}

Result:

0,1,2,3,4,5,6,7,8,9,

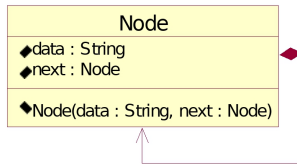
Unidirectional Linked List

Unidirectional Linked List Single Link:

Is a chained storage structure of a linear table, which is connected by a node. Each node consists of data and next pointer to the next node.



UML Diagram

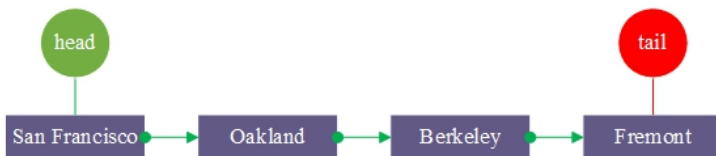


Node.cs

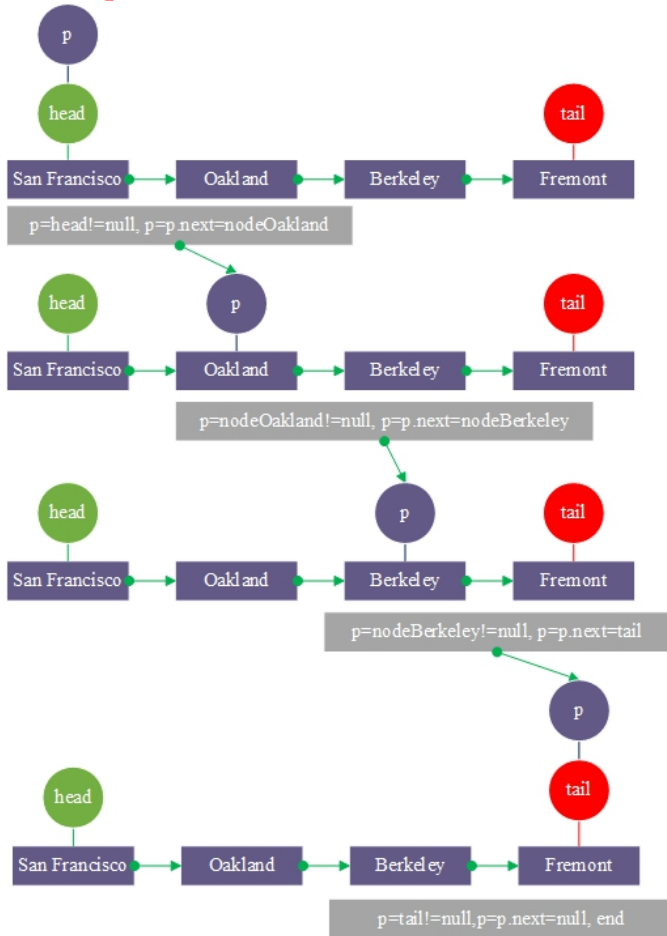
```
namespace CSharpDatastructuresAlgorithms
{
    public class Node
    {
        public String data;
        public Node next;
        public Node(String data, Node next)
        {
            this.data = data;
            this.next = next;
        }
    }
}
```

1. Unidirectional Linked List **initialization** .

Example : Construct a San Francisco subway Unidirectional linked list



2. traversal output .



TestUnidirectionalLinkedList.cs

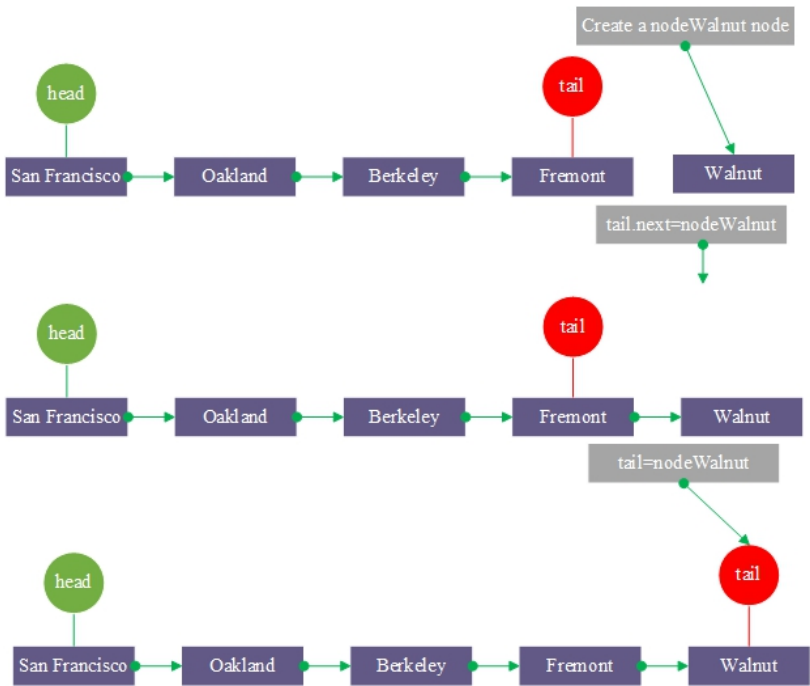
```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestUnidirectionalLinkedList
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Print(head);
        }
        public static Node Init()
        {
            // the first node called head node
            head = new Node ("San Francisco" , null );
            Node nodeOakland = new Node ("Oakland" , null );
            head.next = nodeOakland;
            Node nodeBerkeley = new Node ("Berkeley" , null );
            nodeOakland.next = nodeBerkeley;
            // the last node called tail node
            tail = new Node ("Fremont" , null );
            nodeBerkeley.next = tail;
            return head;
        }
        public static void Print(Node node)
        {
            Node p = node;
            while (p != null ) // From the beginning to the end
            {
                Debug .Write(p.data + " -> " );
                p = p.next;
            }
            Debug .Write("End\n\n");
        }
    }
}
```

}
}

Result:

San Francisco -> Oakland -> Berkeley -> Fremont -> End

3. Append a new node name: **Walnut** to the end.



TestUnidirectionalLinkedList.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestUnidirectionalLinkedList
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Debug.WriteLine("Append a new node name: Walnut to the
end: \n");
            Add(new Node ("Walnut" , null ));
            Print(head);
        }
        public static Node Init()
        {
            // the first node called head node
            head = new Node ("San Francisco" , null );
            Node nodeOakland = new Node ("Oakland" , null );
            head.next = nodeOakland;
            Node nodeBerkeley = new Node ("Berkeley" , null );
            nodeOakland.next = nodeBerkeley;
            // the last node called tail node
            tail = new Node ("Fremont" , null );
            nodeBerkeley.next = tail;
            return head;
        }
        public static void Add(Node newNode)
        {
            tail.next = newNode;
            tail = newNode;
        }
        public static void Print(Node node)
        {

```

```

Node p = node;
while (p != null ) // From the beginning to the end
{
    Debug .Write(p.data + " -> ");
    p = p.next;
}
Debug .Write("End\n\n");
}
}
}

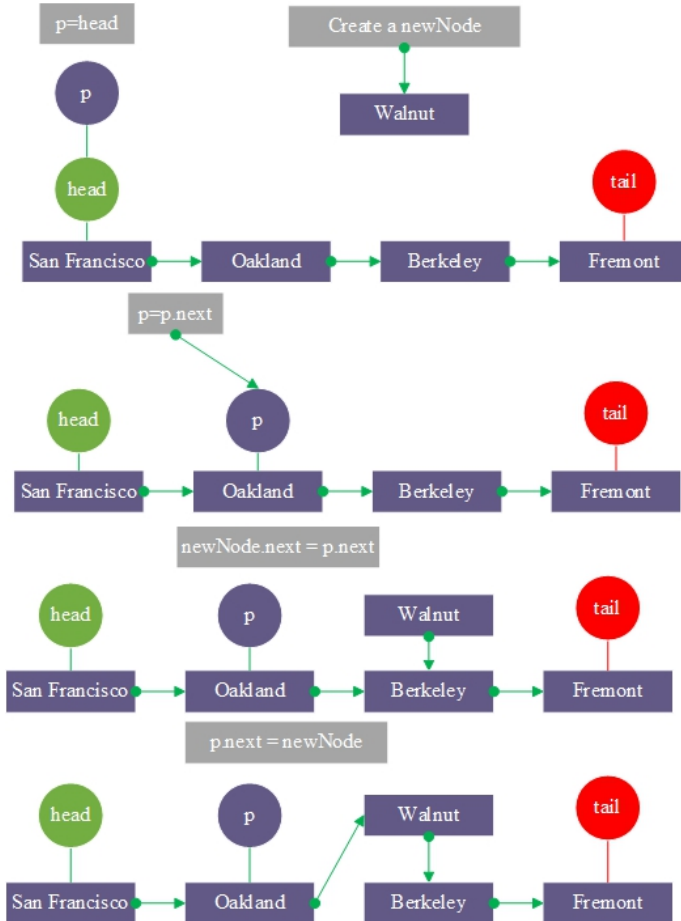
```

Result:

Append a new node name: Walnut to the end:

San Francisco -> Oakland -> Berkeley -> Fremont -> Walnut ->
End

3. Insert a node **Walnut** in position 2.



TestUnidirectionalLinkedList.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestUnidirectionalLinkedList
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Debug.WriteLine("Insert a new node Walnut at index = 2 :
\n" );
            Insert(2, new Node ("Walnut" , null ));
            Print(head);
        }
        public static Node Init()
        {
            // the first node called head node
            head = new Node ("San Francisco" , null );
            Node nodeOakland = new Node ("Oakland" , null );
            head.next = nodeOakland;
            Node nodeBerkeley = new Node ("Berkeley" , null );
            nodeOakland.next = nodeBerkeley;
            // the last node called tail node
            tail = new Node ("Fremont" , null );
            nodeBerkeley.next = tail;
            return head;
        }
        public static void Insert(int insertPosition, Node newNode)
        {
            Node p = head;
            int i = 0;
            // Move the node to the insertion position
            while (p.next != null && i < insertPosition - 1)
            {
```

```

        p = p.next;
        i++;
    }
    newNode.next = p.next; // newNode next point to next
node
    p.next = newNode; // current next point to newNode
}
public static void Print(Node node)
{
    Node p = node;
    while (p != null ) // From the beginning to the end
    {
        Debug.WriteLine(p.data + " -> ");
        p = p.next;
    }
    Debug.WriteLine("End\n\n");
}
}
}

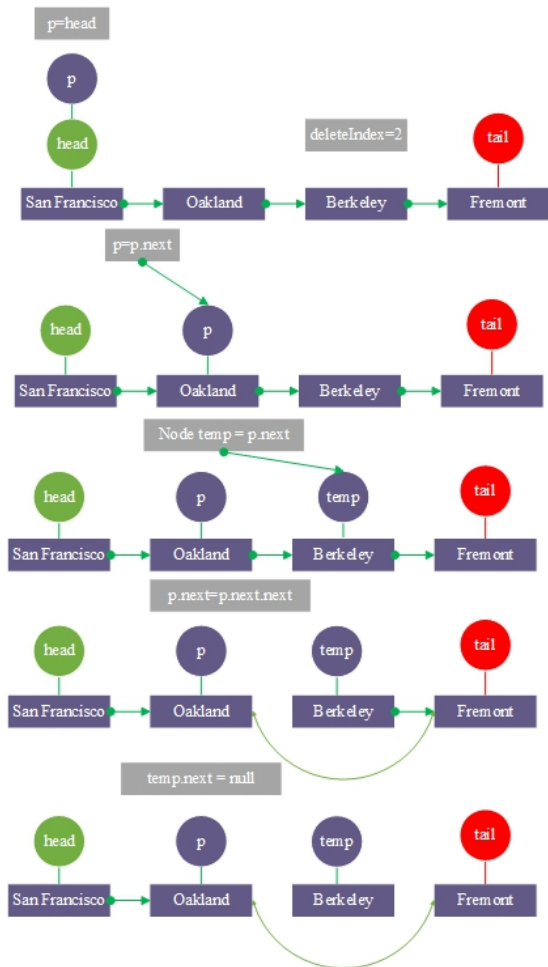
```

Result:

Insert a new node Walnut at index = 2 :

San Francisco -> Oakland -> Walnut -> Berkeley -> Fremont ->
End

4. Delete the **index = 2** node.



TestUnidirectionalLinkedList.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestUnidirectionalLinkedList
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Debug.WriteLine("Delete a new node Berkeley at index = 2 :
\n" );
            Remove(2);
            Print(head);
        }
        public static Node Init()
        {
            // the first node called head node
            head = new Node ("San Francisco" , null );
            Node nodeOakland = new Node ("Oakland" , null );
            head.next = nodeOakland;
            Node nodeBerkeley = new Node ("Berkeley" , null );
            nodeOakland.next = nodeBerkeley;
            // the last node called tail node
            tail = new Node ("Fremont" , null );
            nodeBerkeley.next = tail;
            return head;
        }
        public static void Remove(int removePosition)
        {
            Node p = head;
            int i = 0;
            // Move the node to the previous node position that was
deleted
            while (p.next != null && i < removePosition - 1)
```

```

    {
        p = p.next;
        i++;
    }
    Node temp = p.next; // Save the node you want to delete
    p.next = p.next.next; // Previous node next points to
next of delete the node
    temp.next = null;
}
public static void Print(Node node)
{
    Node p = node;
    while (p != null) // From the beginning to the end
    {
        Debug.WriteLine(p.data + " -> ");
        p = p.next;
    }
    Debug.WriteLine("End\n\n");
}
}
}

```

Result:

Delete a new node Berkeley at index = 2 :
 San Francisco -> Oakland -> Fremont -> End

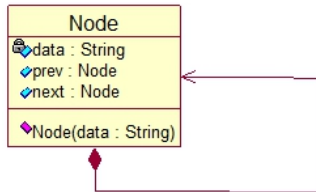
Doubly Linked List

Doubly Linked List:

It is a chained storage structure of a linear table. It is connected by nodes in two directions. Each node consists of data, pointing to the previous node and pointing to the next node.



UML Diagram

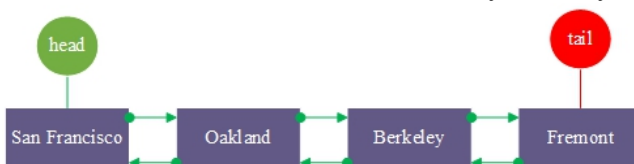


Node.cs

```
namespace CSharpDatastructuresAlgorithms
{
    public class Node
    {
        public String data;
        public Node prev;
        public Node next;
        public Node(String data)
        {
            this.data = data;
        }
    }
}
```

1. Doubly Linked List **initialization** .

Example : Construct a San Francisco subway Doubly linked list



2. traversal output . TestDoubleLink.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestDoubleLink
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Print(head);
        }
        public static void Init()
        {
            head = new Node ("San Francisco" );
            head.prev = null ;
            head.next = null ;
            Node nodeOakland = new Node ("Oakland" );
            nodeOakland.prev = head;
            nodeOakland.next = null ;
            head.next = nodeOakland;
            Node nodeBerkeley = new Node ("Berkeley" );
            nodeBerkeley.prev = nodeOakland;
            nodeBerkeley.next = null ;
            nodeOakland.next = nodeBerkeley;
            tail = new Node ("Fremont" );
            tail.prev = nodeBerkeley;
            tail.next = null ;
            nodeBerkeley.next = tail;
        }
        public static void Print(Node node)
        {
            Node p = node;
            Node end = null ;
            while (p != null )
```

```

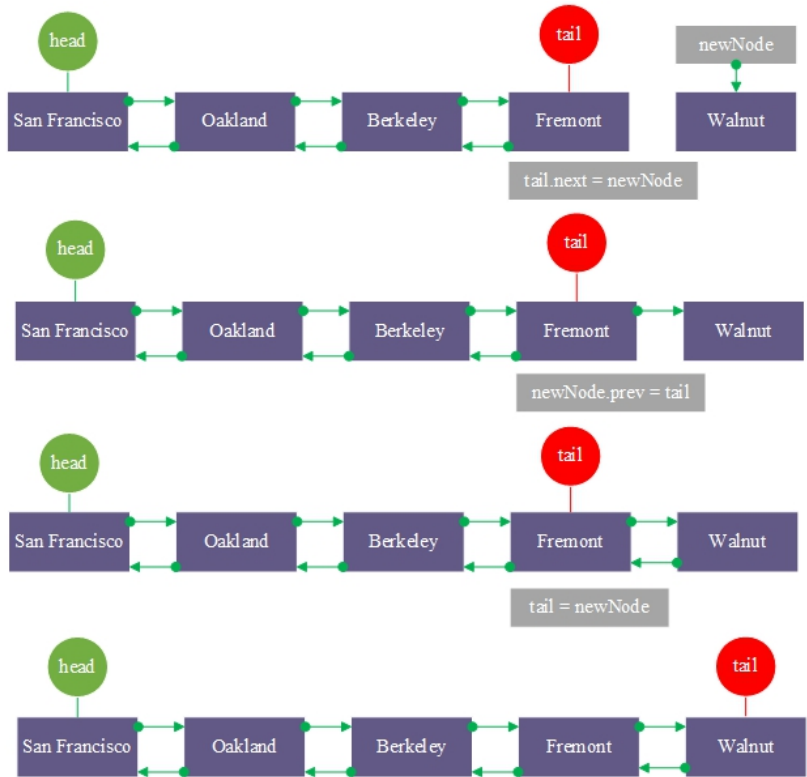
{
    Debug .Write(p.data + " -> ");
    end = p;
    p = p.next;
}
Debug .Write("End\n");
p = end;
while (p != null )
{
    Debug .Write(p.data + " -> ");
    p = p.prev;
}
Debug .Write("Start\n\n");
}
}
}

```

Result:

San Francisco -> Oakland -> Berkeley -> Fremont -> End
 Fremont -> Berkeley -> Oakland -> San Francisco -> Start

3. add a node **Walnut** at the end of Fremont.



TestDoubleLink.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestDoubleLink
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Add(new Node ("Walnut" ));
            Print(head);
        }
        public static void Init()
        {
            head = new Node ("San Francisco" );
            head.prev = null ;
            head.next = null ;
            Node nodeOakland = new Node ("Oakland" );
            nodeOakland.prev = head;
            nodeOakland.next = null ;
            head.next = nodeOakland;
            Node nodeBerkeley = new Node ("Berkeley" );
            nodeBerkeley.prev = nodeOakland;
            nodeBerkeley.next = null ;
            nodeOakland.next = nodeBerkeley;
            tail = new Node ("Fremont" );
            tail.prev = nodeBerkeley;
            tail.next = null ;
            nodeBerkeley.next = tail;
        }
        public static void Add(Node newNode)
        {
            tail.next = newNode;
            newNode.prev = tail;
```

```

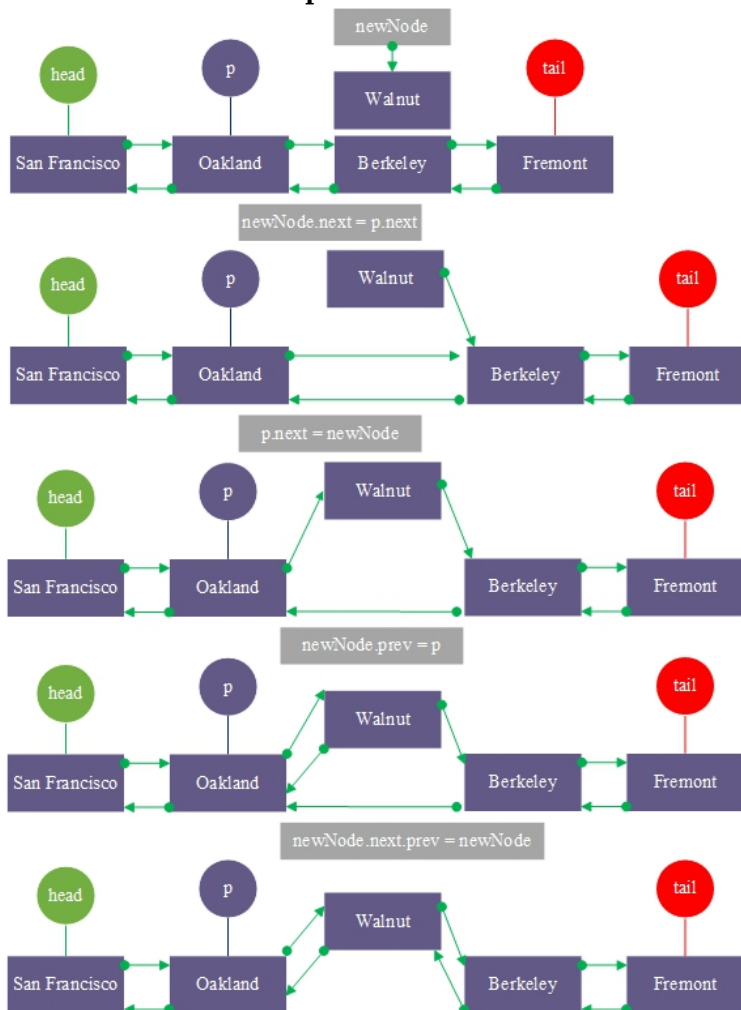
        tail = newNode;
    }
    public static void Print(Node node)
    {
        Node p = node;
        Node end = null ;
        while (p != null )
        {
            Debug .Write(p.data + " -> ");
            end = p;
            p = p.next;
        }
        Debug .Write("End\n");
        p = end;
        while (p != null )
        {
            Debug .Write(p.data + " -> ");
            p = p.prev;
        }
        Debug .Write("Start\n\n");
    }
}

```

Result:

San Francisco -> Oakland -> Berkeley -> Fremont -> Walnut ->
 End
 Walnut -> Fremont -> Berkeley -> Oakland -> San Francisco ->
 Start

3. Insert a node **Walnut** in position 2.



TestDoubleLink.cs

```
using System;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestDoubleLink
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Insert(2, new Node ("Walnut" ));
            Print(head);
        }
        public static void Init()
        {
            head = new Node ("San Francisco" );
            head.prev = null ;
            head.next = null ;
            Node nodeOakland = new Node ("Oakland" );
            nodeOakland.prev = head;
            nodeOakland.next = null ;
            head.next = nodeOakland;
            Node nodeBerkeley = new Node ("Berkeley" );
            nodeBerkeley.prev = nodeOakland;
            nodeBerkeley.next = null ;
            nodeOakland.next = nodeBerkeley;
            tail = new Node ("Fremont" );
            tail.prev = nodeBerkeley;
            tail.next = null ;
            nodeBerkeley.next = tail;
        }
        public static void Insert(int insertPosition, Node newNode)
        {
            Node p = head;
            int i = 0;
            while (p.next != null && i < insertPosition - 1)
```

```

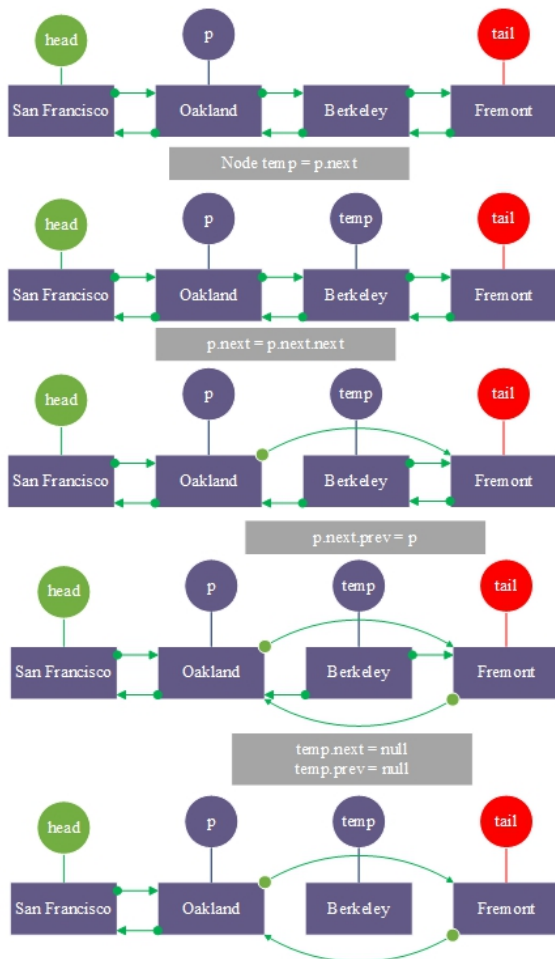
    {
        p = p.next;
        i ++;
    }
    newNode.next = p.next;
    p.next = newNode;
    newNode.prev = p;
    newNode.next.prev = newNode;
}
public static void Print(Node node)
{
    Node p = node;
    Node end = null ;
    while (p != null )
    {
        Debug .Write(p.data + " -> ");
        end = p;
        p = p.next;
    }
    Debug .Write("End\n");
    p = end;
    while (p != null )
    {
        Debug .Write(p.data + " -> ");
        p = p.prev;
    }
    Debug .Write("Start\n\n");
}
}
}

```

Result:

San Francisco -> Oakland -> Walnut -> Berkeley -> Fremont ->
End
Fremont -> Berkeley -> Walnut -> Oakland -> San Francisco ->
Start

4. Delete the **index = 2** node.



TestDoubleLink.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestDoubleLink
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Remove(2);
            Print(head);
        }
        public static void Init()
        {
            head = new Node ("San Francisco" );
            head.prev = null ;
            head.next = null ;
            Node nodeOakland = new Node ("Oakland" );
            nodeOakland.prev = head;
            nodeOakland.next = null ;
            head.next = nodeOakland;
            Node nodeBerkeley = new Node ("Berkeley" );
            nodeBerkeley.prev = nodeOakland;
            nodeBerkeley.next = null ;
            nodeOakland.next = nodeBerkeley;
            tail = new Node ("Fremont" );
            tail.prev = nodeBerkeley;
            tail.next = null ;
            nodeBerkeley.next = tail;
        }
        public static void Remove(int removePosition)
        {
            Node p = head;
            int i = 0;
```



```

// Move the node to the previous node that was deleted
while (p.next != null && i < removePosition - 1)
{
    p = p.next;
    i++;
}
Node temp = p.next; // Save the node you want to delete
p.next = p.next.next;
p.next.prev = p;
temp.next = null; // Set the delete node next to null
temp.prev = null; // Set the delete node prev to null
}
public static void Print(Node node)
{
    Node p = node;
    Node end = null;
    while (p != null)
    {
        Debug.WriteLine(p.data + " -> ");
        end = p;
        p = p.next;
    }
    Debug.WriteLine("End\n");
    p = end;
    while (p != null)
    {
        Debug.WriteLine(p.data + " -> ");
        p = p.prev;
    }
    Debug.WriteLine("Start\n\n");
}
}
}

```

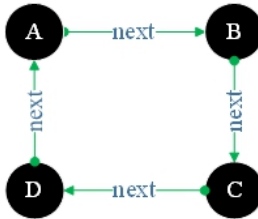
Result:

San Francisco -> Oakland -> Fremont -> End
 Fremont -> Oakland -> San Francisco -> Start

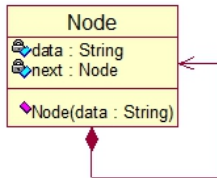
One-way Circular LinkedList

One-way Circular List:

It is a chain storage structure of a linear table, which is connected to form a ring, and each node is composed of data and a pointer to next.



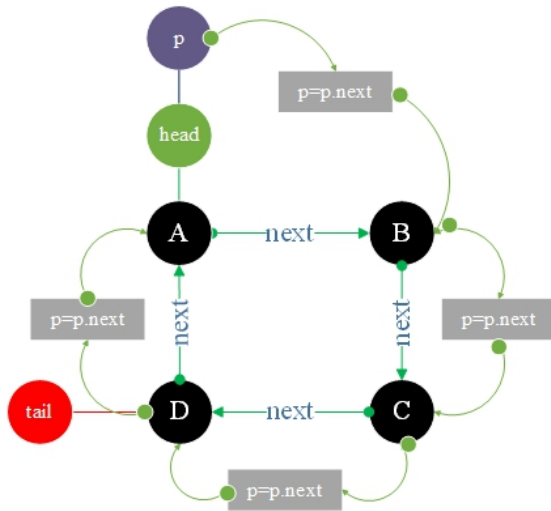
UML Diagram



Node.cs

```
namespace CSharpDatastructuresAlgorithms
{
    public class Node
    {
        public String data;
        public Node next;
        public Node(String data)
        {
            this.data = data;
        }
    }
}
```

1. One-way Circular Linked List **initialization and traversal** output .



TestSingleCircleLink.cs

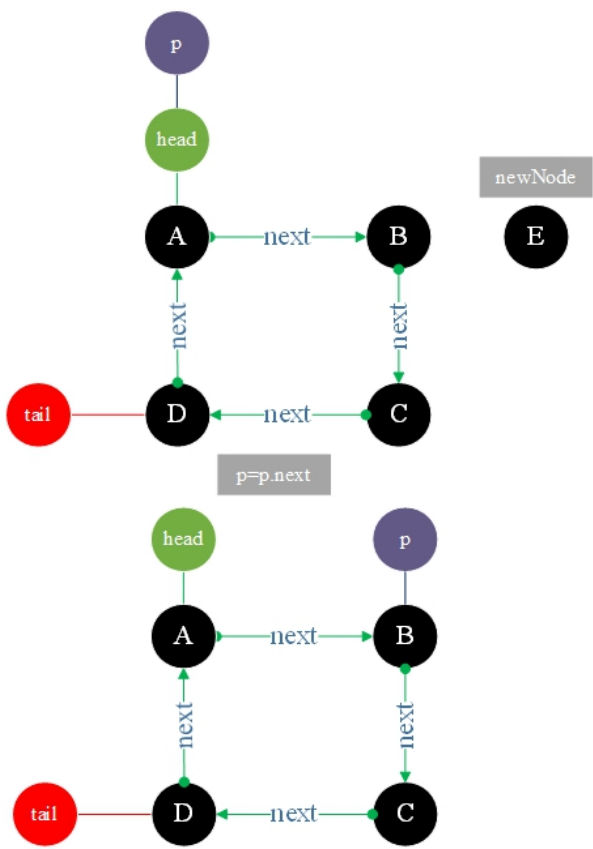
```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestSingleCircleLink
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Print();
        }
        public static void Init()
        {
            // the first node called head node
            head = new Node ("A" );
            head.next = null ;
            Node nodeB = new Node ("B" );
            nodeB.next = null ;
            head.next = nodeB;
            Node nodeC = new Node ("C" );
            nodeC.next = null ;
            nodeB.next = nodeC;
            // the last node called tail node
            tail = new Node ("D" );
            tail.next = head;
            nodeC.next = tail;
        }
        public static void Print()
        {
            Node p = head;
            do
            {
                Debug .Write(p.data + " -> " );
                p = p.next;
            }
            while (p != head);
        }
    }
}
```

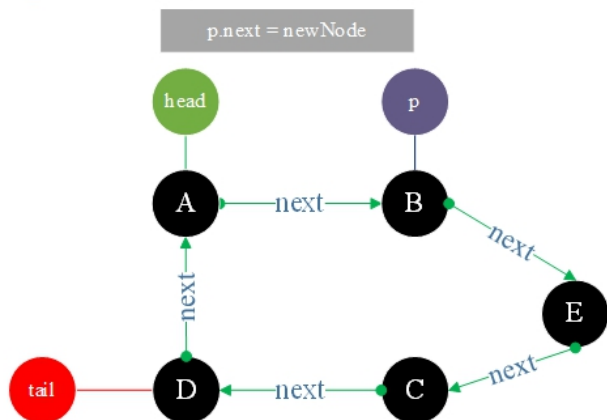
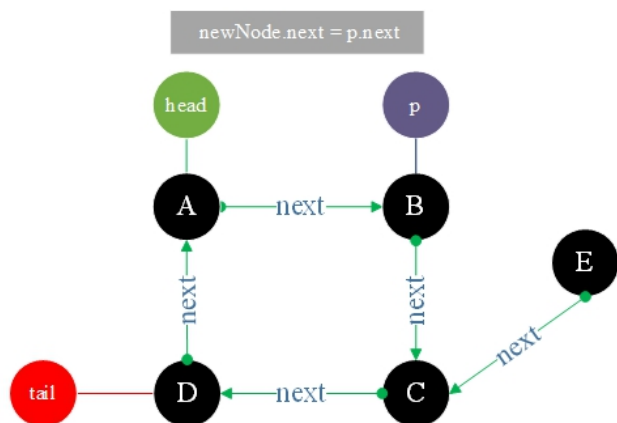
```
        } while (p != head);  
        Debug.WriteLine(p.data + "\n\n");  
    }  
}
```

Result:

A -> B -> C -> D -> A

3. Insert a node **E** in position 2.





TestSingleCircleLink.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestSingleCircleLink
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Insert(2, new Node ("E" ));
            Print();
        }
        public static void Init()
        {
            // the first node called head node
            head = new Node ("A" );
            head.next = null ;
            Node nodeB = new Node ("B" );
            nodeB.next = null ;
            head.next = nodeB;
            Node nodeC = new Node ("C" );
            nodeC.next = null ;
            nodeB.next = nodeC;
            // the last node called tail node
            tail = new Node ("D" );
            tail.next = head;
            nodeC.next = tail;
        }
        public static void Insert(int insertPosition, Node newNode)
        {
            Node p = head;
            int i = 0;
            // Move the node to the insertion position
            while (p.next != null && i < insertPosition - 1)
```



```

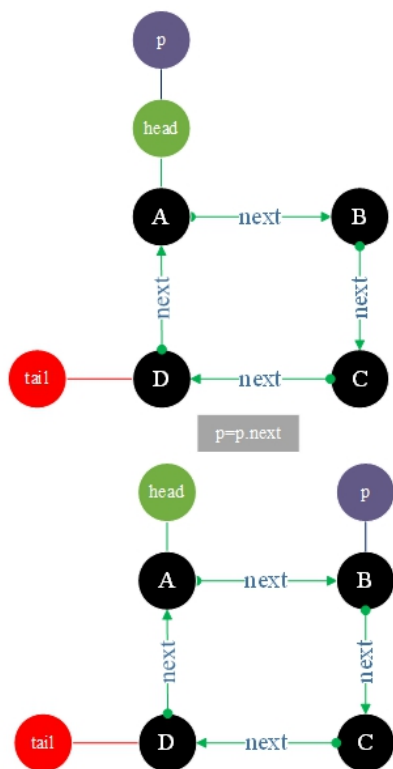
    {
        p = p.next;
        i++;
    }
    newNode.next = p.next;
    p.next = newNode;
}
public static void Print()
{
    Node p = head;
    do
    {
        Debug.WriteLine(p.data + " -> ");
        p = p.next;
    } while (p != head);
    Debug.WriteLine(p.data + "\n\n");
}
}
}

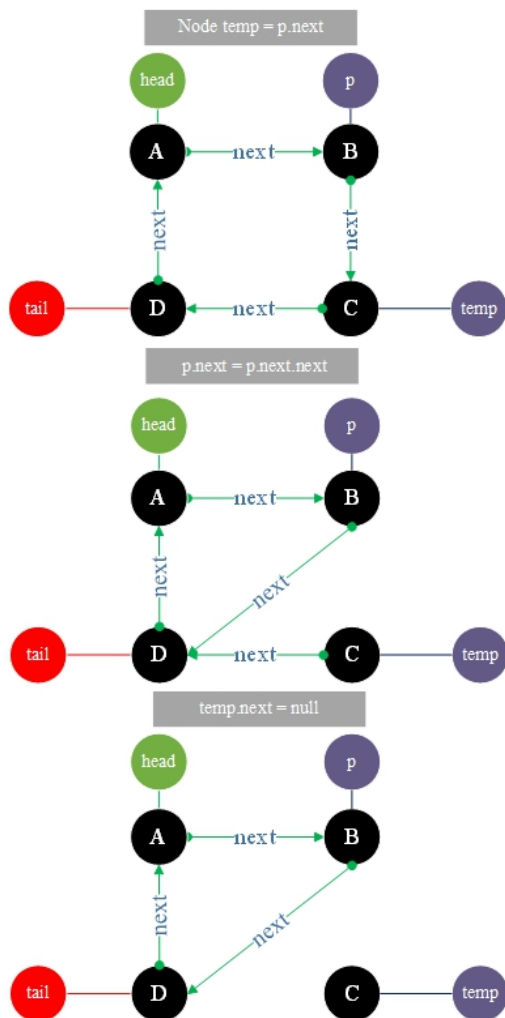
```

Result:

A -> B -> E -> C -> D -> A

4. Delete the **index = 2** node.





TestSingleCircleLink.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestSingleCircleLink
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Remove(2);
            Print();
        }
        public static void Init()
        {
            // the first node called head node
            head = new Node ("A" );
            head.next = null ;
            Node nodeB = new Node ("B" );
            nodeB.next = null ;
            head.next = nodeB;
            Node nodeC = new Node ("C" );
            nodeC.next = null ;
            nodeB.next = nodeC;
            // the last node called tail node
            tail = new Node ("D" );
            tail.next = head;
            nodeC.next = tail;
        }
        public static void Remove(int removePosition)
        {
            Node p = head;
            int i = 0;
            while (p.next != null && i < removePosition - 1)
            {
```

```

        p = p.next;
        i++;
    }
    Node temp = p.next;
    p.next = p.next.next;
    temp.next = null;
}
public static void Print()
{
    Node p = head;
    do
    {
        Debug.WriteLine(p.data + " -> ");
        p = p.next;
    } while (p != head);
    Debug.WriteLine(p.data + "\n\n");
}
}
}

```

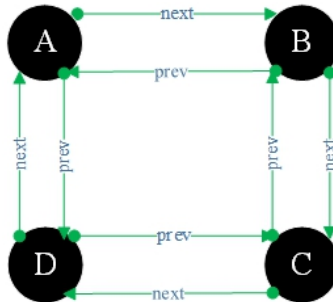
Result:

A -> B -> D -> A

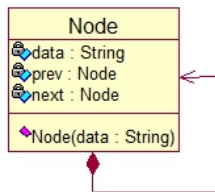
Two-way Circular LinkedList

Two-way Circular List:

It is a chain storage structure of a linear table. The nodes are connected in series by two directions, and is connected to form a ring. Each node is composed of **data**, pointing to the previous node **prev** and pointing to the next node **next**.



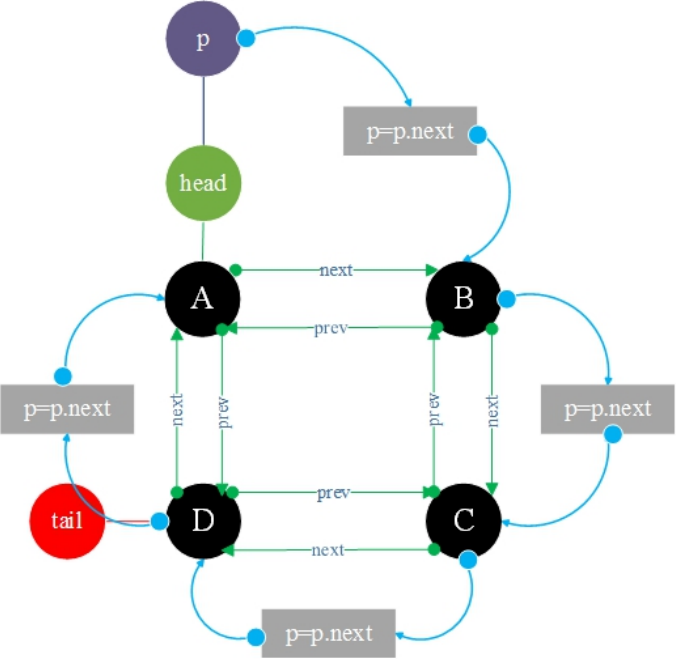
UML Diagram

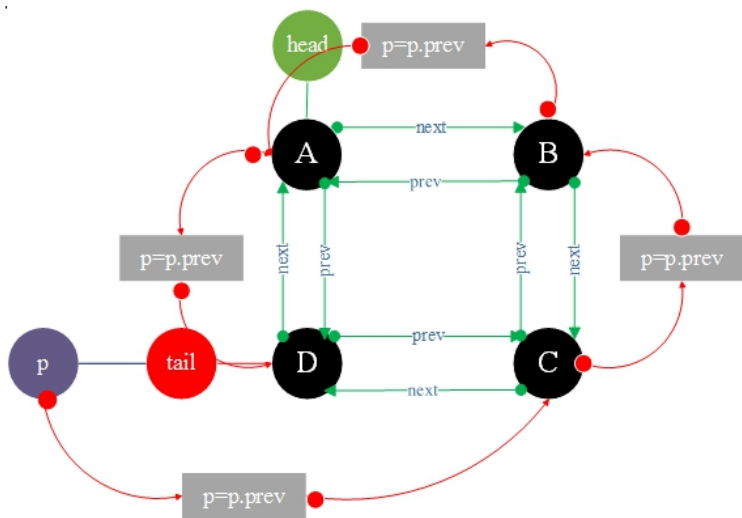


Node.cs

```
namespace CSharpDatastructuresAlgorithms
{
    public class Node
    {
        public String data;
        public Node prev;
        public Node next;
        public Node(String data)
        {
            this.data = data;
        }
    }
}
```

1. Two-way Circular Linked List **initialization and traversal** output .





TestDoubleCircleLink.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestDoubleCircleLink
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Print();
        }
        public static void Init()
        {
            // the first node called head node
            head = new Node ("A" );
            head.next = null ;
            head.prev = null ;
            Node nodeB = new Node ("B" );
            nodeB.next = null ;
            nodeB.prev = head;
            head.next = nodeB;
            Node nodeC = new Node ("C" );
            nodeC.next = null ;
            nodeC.prev = nodeB;
            nodeB.next = nodeC;
            // the last node called tail node
            tail = new Node ("D" );
            tail.next = head;
            tail.prev = nodeC;
            nodeC.next = tail;
            head.prev = tail;
        }
        public static void Print()
        {

```

```

Node p = head;
do
{
    Debug .Write(p.data + " -> ");
    p = p.next;
} while (p != head);
Debug .Write(p.data + "\n\n");
p = tail;
do
{
    Debug .Write(p.data + " -> ");
    p = p.prev;
} while (p != tail);
Debug .Write(p.data + "\n\n");
}
}
}

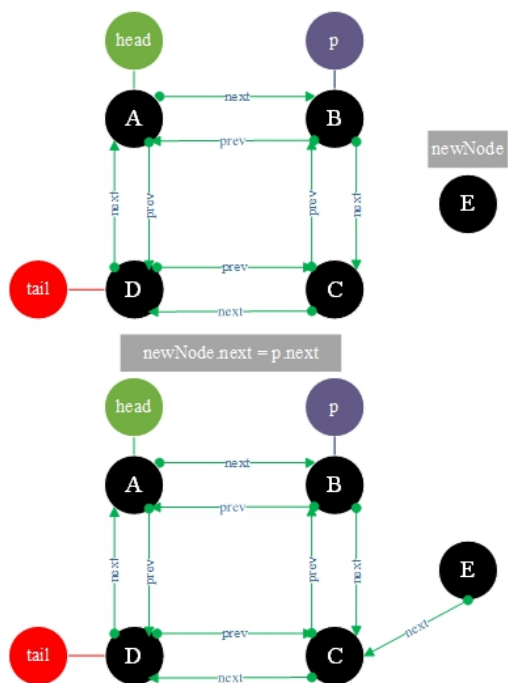
```

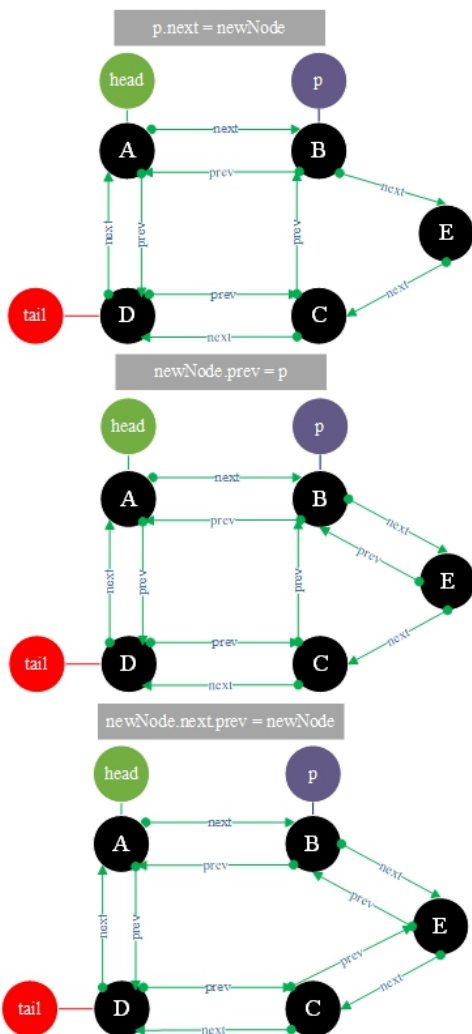
Result:

A -> B -> C -> D -> A

D -> C -> B -> A -> D

3. Insert a node **E** in position 2.





TestDoubleCircleLink.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestDoubleCircleLink
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Insert(2, new Node ("E" ));
            Print();
        }
        public static void Init()
        {
            head = new Node ("A" );
            head.next = null ;
            head.prev = null ;
            Node nodeB = new Node ("B" );
            nodeB.next = null ;
            nodeB.prev = head;
            head.next = nodeB;
            Node nodeC = new Node ("C" );
            nodeC.next = null ;
            nodeC.prev = nodeB;
            nodeB.next = nodeC;
            tail = new Node ("D" );
            tail.next = head;
            tail.prev = nodeC;
            nodeC.next = tail;
            head.prev = tail;
        }
        public static void Insert(int insertPosition, Node newNode)
        {
            Node p = head;
```

```

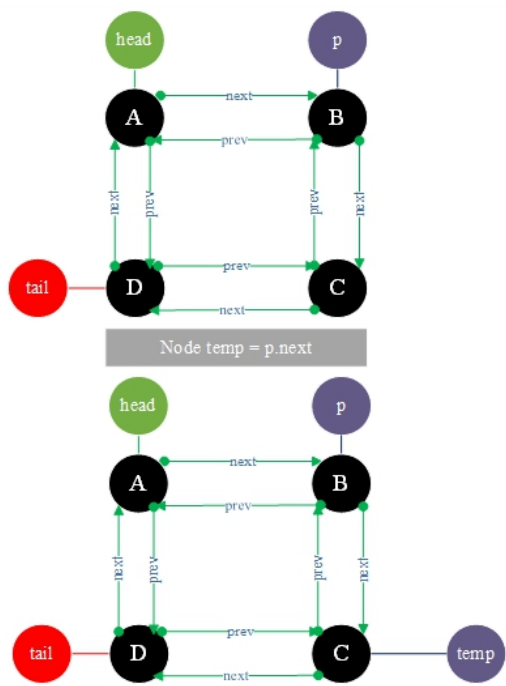
int i = 0;
//Move the node to the insertion position
while (p.next != null && i < insertPosition - 1)
{
    p = p.next;
    i++;
}
newNode.next = p.next;
p.next = newNode;
newNode.prev = p;
newNode.next.prev = newNode;
}
public static void Print()
{
    Node p = head;
    do
    {
        Debug.WriteLine(p.data + " -> ");
        p = p.next;
    } while (p != head);
    Debug.WriteLine(p.data + "\n\n");
    p = tail;
    do
    {
        Debug.WriteLine(p.data + " -> ");
        p = p.prev;
    } while (p != tail);
    Debug.WriteLine(p.data + "\n\n");
}
}
}

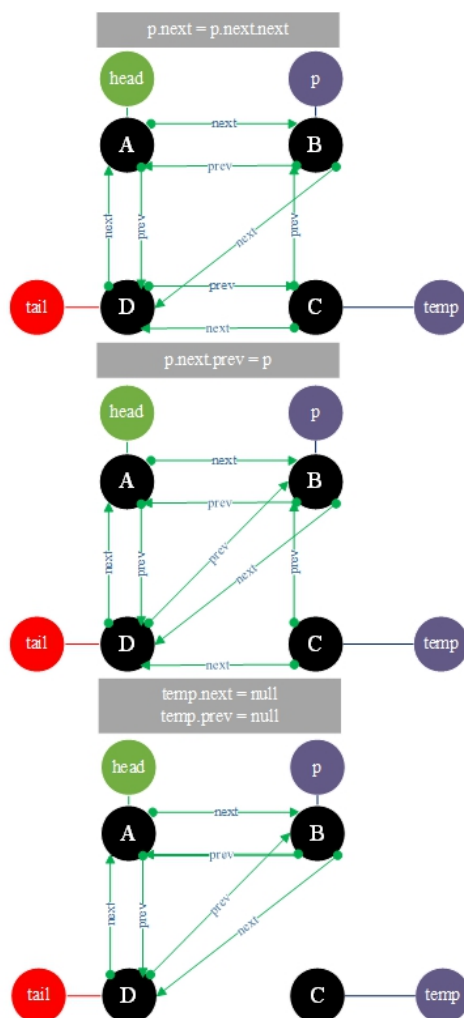
```

Result:

A -> B -> E -> C -> D -> A

4. Delete the **index = 2** node.





TestDoubleCircleLink.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestDoubleCircleLink
    {
        private static Node head;
        private static Node tail;
        public static void Main(String [] args)
        {
            Init();
            Remove(2);
            Print();
        }
        public static void Init()
        {
            head = new Node ("A" );
            head.next = null ;
            head.prev = null ;
            Node nodeB = new Node ("B" );
            nodeB.next = null ;
            nodeB.prev = head;
            head.next = nodeB;
            Node nodeC = new Node ("C" );
            nodeC.next = null ;
            nodeC.prev = nodeB;
            nodeB.next = nodeC;
            tail = new Node ("D" );
            tail.next = head;
            tail.prev = nodeC;
            nodeC.next = tail;
            head.prev = tail;
        }
        public static void Remove(int removePosition)
        {
            Node p = head;
```

```

int i = 0;
while (p.next != null && i < removePosition - 1)
{
    p = p.next;
    i++;
}
Node temp = p.next;
p.next = p.next.next;
p.next.prev = p;
temp.next = null ;//Set the delete node next to null
temp.prev = null ;// Set the delete node prev to null
}
public static void Print()
{
    Node p = head;
    do
    {
        Debug .Write(p.data + " -> ");
        p = p.next;
    } while (p != head);
    Debug .Write(p.data + "\n\n");
    p = tail;
    do
    {
        Debug .Write(p.data + " -> ");
        p = p.prev;
    } while (p != tail);
    Debug .Write(p.data + "\n\n");
}
}
}

```

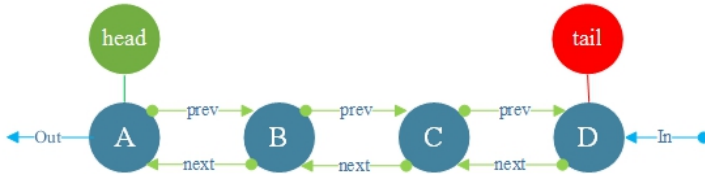
Result:

A -> B -> D -> A

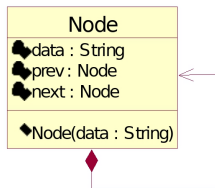
Queue

Queue:

FIFO (First In First Out) sequence.



UML Diagram



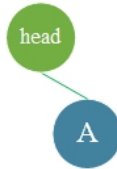
Node.cs

```
namespace CSharpDatastructuresAlgorithms
{
    public class Node
    {
        public String data;
        public Node prev;
        public Node next;
        public Node(String data)
        {
            this.data = data;
        }
    }
}
```

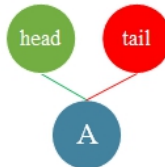
1. Queue **initialization and traversal output** .

Initialization Insert **A**

```
head = new Node( " A " );
```

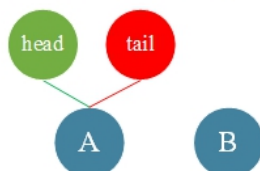


```
tail = head;
```

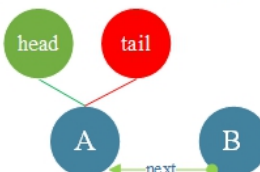


Initialization Insert B

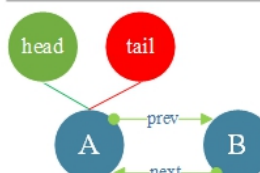
```
newNode = new Node( "B" );
```



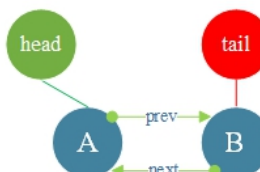
```
newNode.next = tail;
```



```
tail.prev = newNode;
```

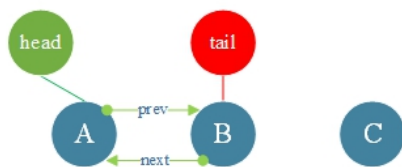


```
tail = newNode;
```

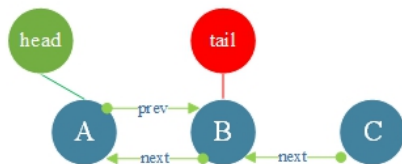


Initialization Insert C

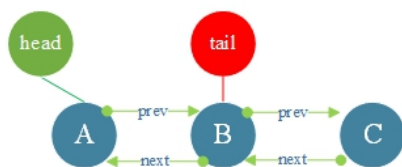
```
newNode = new Node( "C" );
```



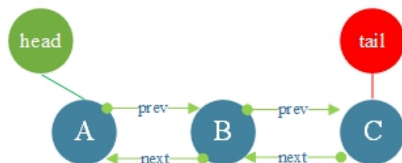
```
newNode.next = tail;
```



```
tail.prev = newNode;
```

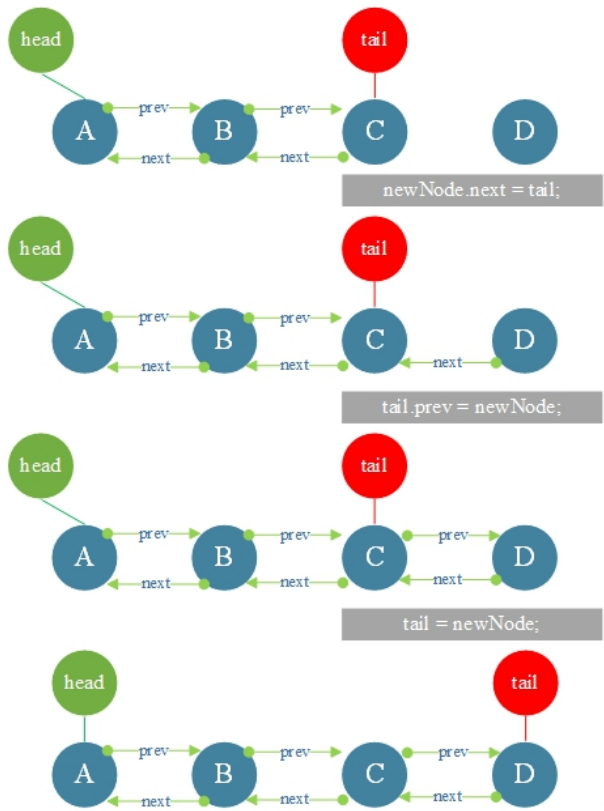


```
tail = newNode;
```



Initialization Insert D

```
newNode = new Node( "D" );
```



TestQueue.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class Queue
    {
        private Node head;
        private Node tail;
        private int size;
        public void Offer(String element)
        {
            if (head == null )
            {
                head = new Node (element);
                tail = head;
            }
            else
            {
                Node newNode = new Node (element);
                newNode.next = tail;
                tail.prev = newNode;
                tail = newNode;
            }
            size ++;
        }
        public Node Poll()
        {
            Node p = head;
            if (p == null )
            {
                return null ;
            }
            head = head.prev;
            p.next = null ;
            p.prev = null ;
            size--;
        }
    }
}
```



```

        return p;
    }
    public int Size()
    {
        return size;
    }
}
class TestQueue
{
    public static void Main(String [] args)
    {
        Queue queue = new Queue ();
        queue.Offer("A" );
        queue.Offer("B" );
        queue.Offer("C" );
        queue.Offer("D" );
        print(queue);
    }
    public static void print(Queue queue)
    {
        Debug .Write("Head " );
        Node node = null ;
        while ((node = queue.Poll()) != null )
        {
            Debug .Write(node.data + " < - " );
        }
        Debug .Write("Tail\n" );
    }
}
}

```

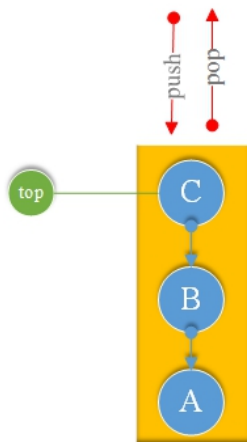
Result:

Head A < - B < - C < - D < - Tail

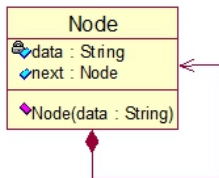
Stack

Stack:

FILO (First In Last Out) sequence.



UML Diagram

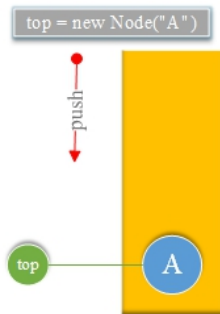


Node.cs

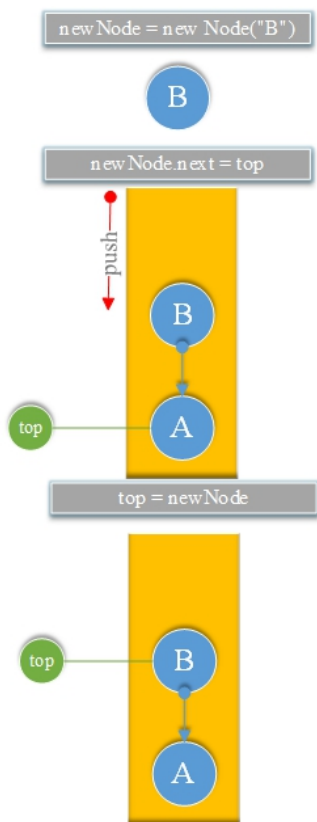
```
namespace CSharpDatastructuresAlgorithms
{
    public class Node
    {
        public String data;
        public Node next;
        public Node(String data)
        {
            this.data = data;
        }
    }
}
```

1. Stack **initialization and traversal output** .

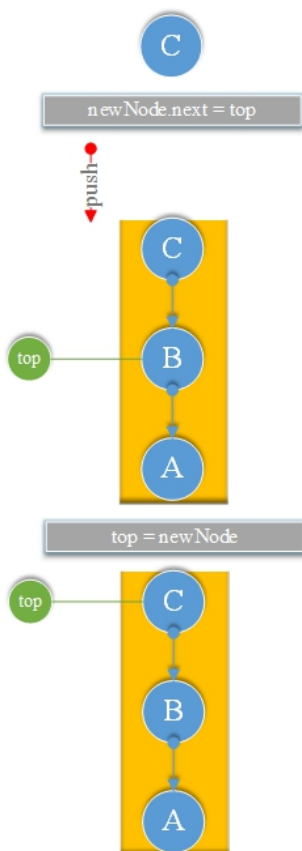
Push **A** into Stack



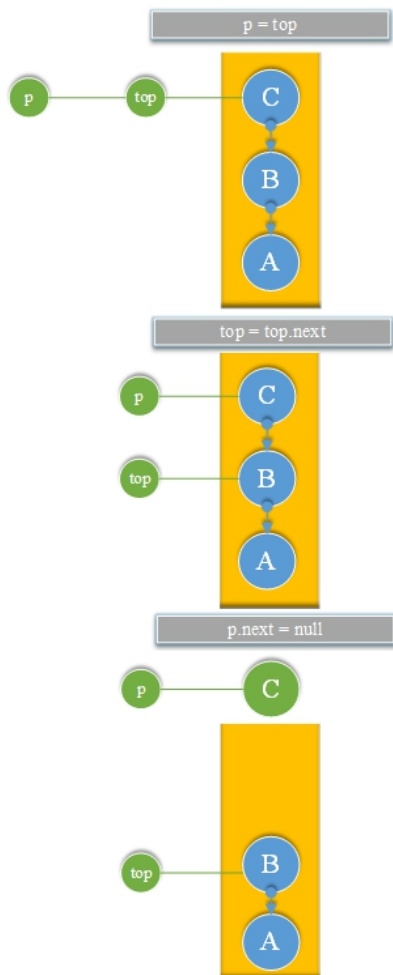
Push **B** into Stack



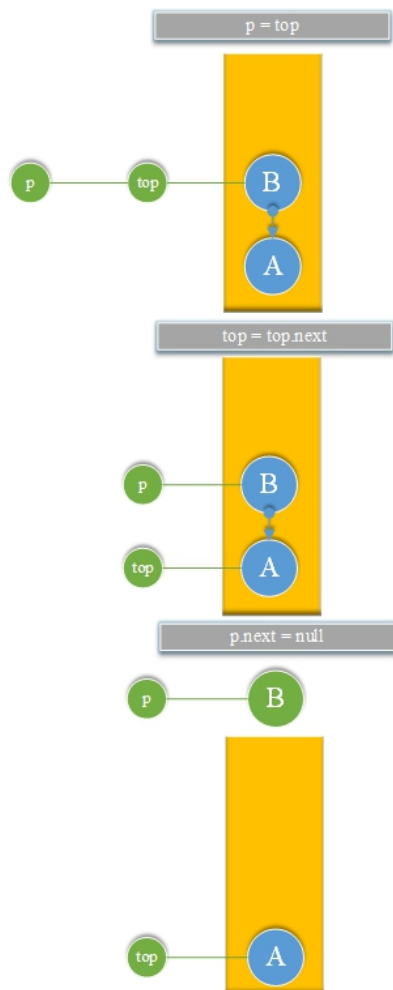
Push C into Stack



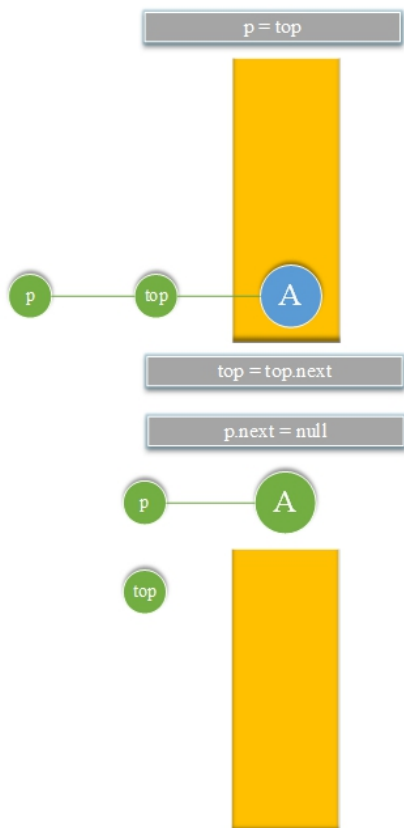
If pop **C** from Stack:



If pop **B** from Stack:



If pop **A** from Stack:



TestStack.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class Stack
    {
        private Node top;
        private int size;
        public void Push(String element)
        {
            if (top == null )
            {
                top = new Node (element);
            }
            else
            {
                Node newNode = new Node (element);
                newNode.next = top;
                top = newNode;
            }
            size ++;
        }
        public Node Pop()
        {
            if (top == null )
            {
                return null ;
            }
            Node p = top;
            top = top.next; // top move down
            p.next = null ;
            size--;
            return p;
        }
        public int Size()
        {
            {
```

```

        return size;
    }
}
class TestStack
{
    public static void Main(String [] args)
    {
        Stack stack = new Stack ();
        stack.Push("A" );
        stack.Push("B" );
        stack.Push("C" );
        stack.Push("D" );
        print(stack);
    }
    public static void print(Stack stack) {
        Debug .Write("Top " );
        Node node = null ;
        while ((node = stack.Pop())!= null ) {
            Debug .Write(node.data + " -> " );
        }
        Debug .Write("End\n" );
    }
}
}

```

Result:

Top D -> C -> B -> A -> End

Recursive Algorithm

Recursive Algorithm:

The program function itself calls its own layer to progress until it reaches a certain condition and step by step returns to the end..

1. Factorial of n : $n*(n-1)*(n-2) \dots *2*1$

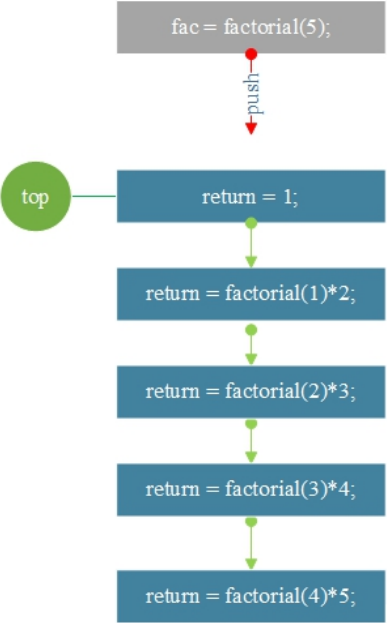
TestFactorial.cs

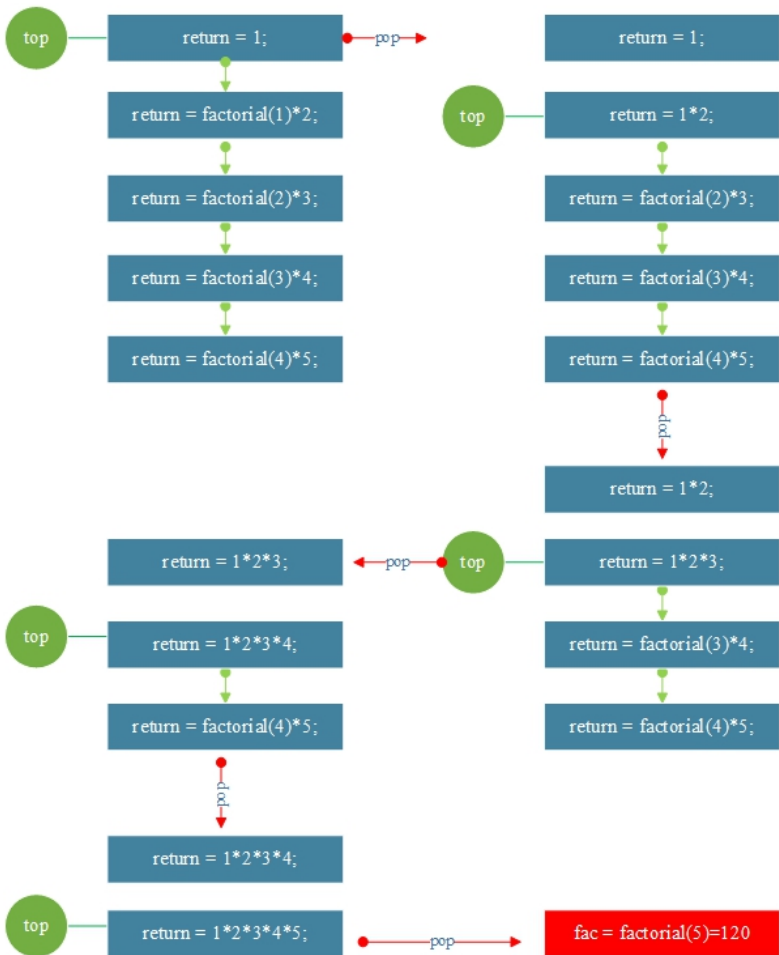
```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestFactorial
    {
        public static void Main(String [] args)
        {
            int n = 5;
            long fac = factorial(n);
            Debug.WriteLine("The factorial of 5 is :" + fac);
        }
        public static long factorial(int n)
        {
            if (n == 1)
            {
                return 1;
            }
            else
            {
                return factorial(n - 1) * n; //Recursively call yourself
            }
        }
    }
}
```

Result:

The factorial of 5 is :120

Graphical analysis:



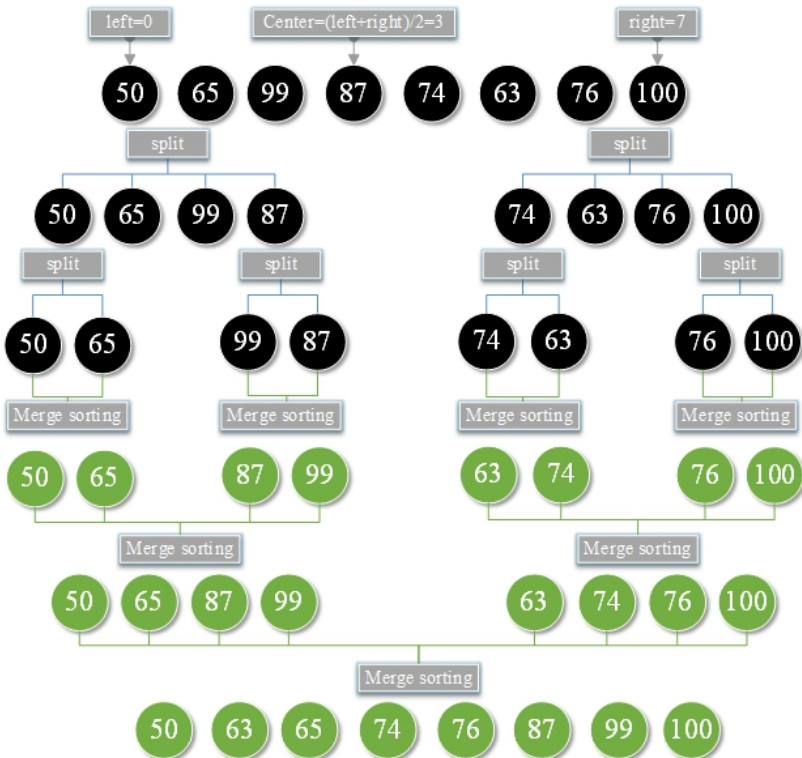


Two-way Merge Algorithm

Two-way Merge Algorithm:

The data of the first half and the second half are sorted, and the two ordered sub-list are merged into one ordered list, which continue to recursive to the end.

1. The scores {50, 65, 99, 87, 74, 63, 76, 100} by merge sort



TestMergeSort.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestMergeSort
    {
        public static void Main(String [] args) {
            int [] scores = { 50, 65, 99, 87, 74, 63, 76, 100, 92 };
            MergeSort(scores);
            for (int i = 0; i < scores.Length; i++) {
                Debug.WriteLine(scores[i] + ",");
            }
        }
        private static void MergeSort(int [] array) {
            MergeSort(array, new int [array.Length], 0, array.Length
- 1);
        }
        private static void MergeSort(int [] array, int [] temp, int
left, int right) {
            if (left < right) {
                int center = (left + right) / 2;
                MergeSort(array, temp, left, center); // Left merge
sort
                MergeSort(array, temp, center + 1, right); // Right
merge sort
                Merge(array, temp, left, center + 1, right); // Merge
two ordered arrays
            }
        }
        /**
         * Combine two ordered list into an ordered list
         * temp : Temporary array
         * left : Start the subscript on the left
         * right : Start the subscript on the right
         * rightEndIndex : End subscript on the right
         */
    }
}
```



```

private static void Merge(int [] array, int [] temp, int left,
int right, int rightEndIndex) {
    int leftEndIndex = right - 1; // End subscript on the left
    int tempIndex = left; // Starting from the left count
    int elementNumber = rightEndIndex - left + 1;
    while (left <= leftEndIndex && right <=
rightEndIndex) {
        if (array[left] <= array[right])
            temp[tempIndex++] = array[left++];
        else
            temp[tempIndex++] = array[right++];
    }
    while (left <= leftEndIndex) { // If there is element on
the left
        temp[tempIndex++] = array[left++];
    }
    while (right <= rightEndIndex) { // If there is element
on the right
        temp[tempIndex++] = array[right++];
    }
    // Copy temp to array
    for (int i = 0; i < elementNumber; i++) {
        array[rightEndIndex] = temp[rightEndIndex];
        rightEndIndex--;
    }
    }
}

```

Result:

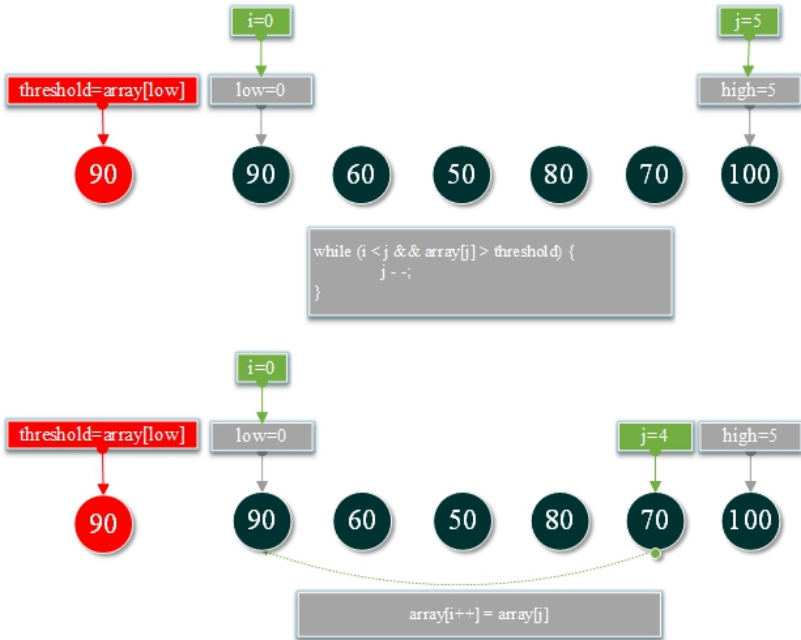
50,63,65,74,76,87,92,99,100,

Quick Sort Algorithm

Quick Sort Algorithm:

Quicksort is a popular sorting algorithm that is often faster in practice compared to other sorting algorithms. It utilizes a divide-and-conquer strategy to quickly sort data items by dividing a large array into two smaller arrays.

1. The scores {90, 60, 50, 80, 70, 100} by quick sort



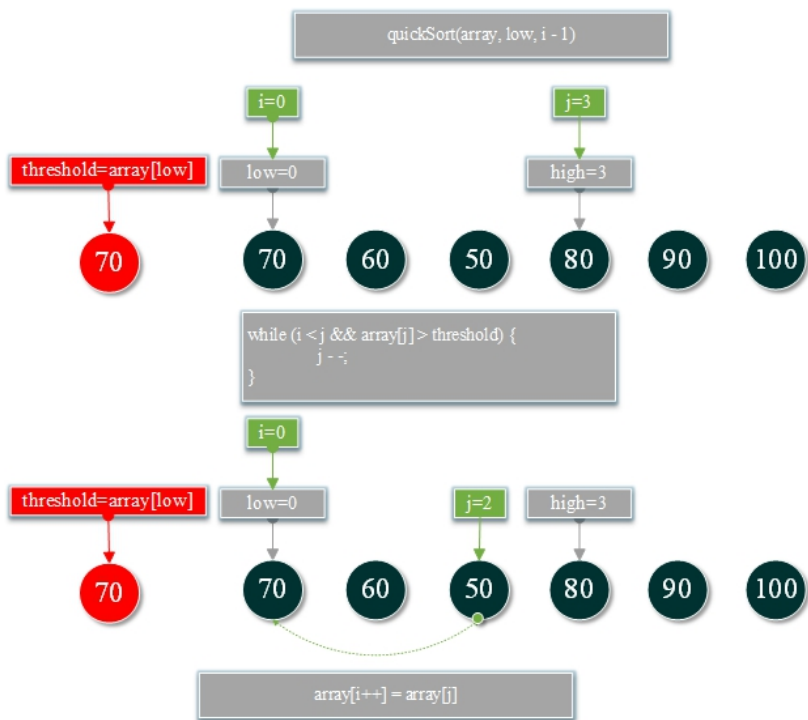


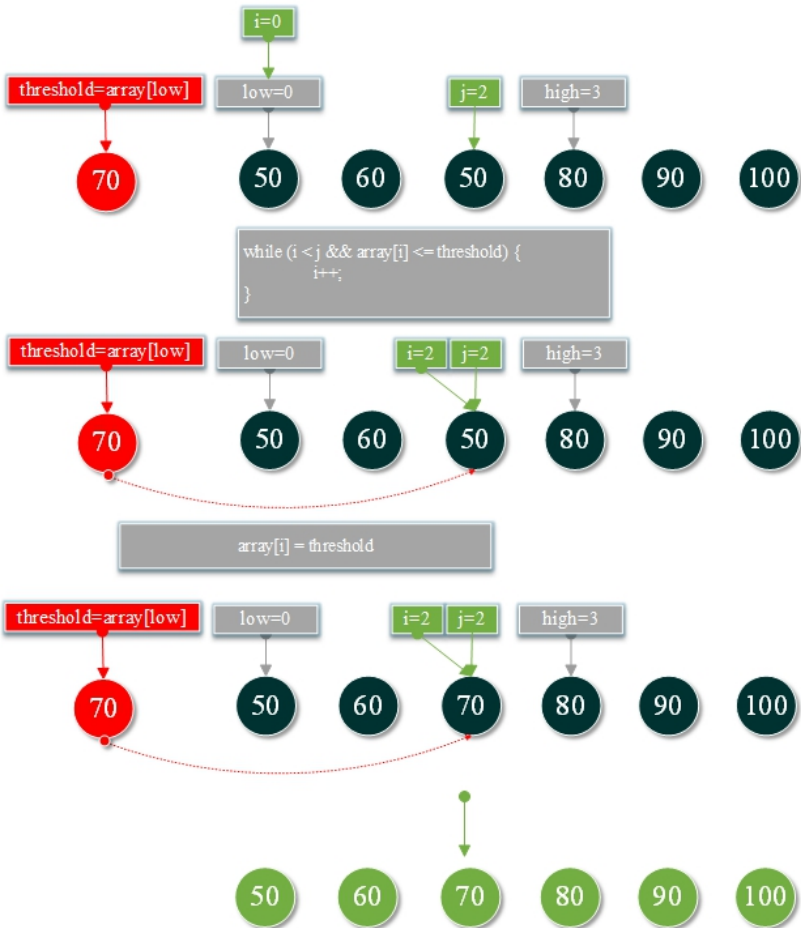
```
while (i < j && array[i] <= threshold) {  
    i++;  
}
```



```
array[i] = threshold
```







TestQuickSort.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestQuickSort
    {
        public static void Main(String [] args)
        {
            int [] scores = { 90, 60, 50, 80, 70, 100 };
            QuickSort(scores);
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.WriteLine(scores[i] + ",");
            }
        }
        public static void QuickSort(int [] array)
        {
            if (array.Length > 0)
            {
                QuickSort(array, 0, array.Length - 1);
            }
        }
        private static void QuickSort(int [] array, int low, int high)
        {
            if (low > high)
            {
                return ;
            }
            int i = low;
            int j = high;
            int threshold = array[low];
            // Alternately scanned from both ends of the list
            while (i < j)
            {
                // Find the first position less than threshold from
                right to left
```

```

        while (i < j && array[j] > threshold)
        {
            j--;
        }
        //Replace the low with a smaller number than the
threshold
        if (i < j)
            array[i + +] = array[j];
        // Find the first position greater than threshold from
left to right
        while (i < j && array[i] <= threshold)
        {
            i + +;
        }
        //Replace the high with a number larger than the
threshold
        if (i < j)
            array[j--] = array[i];
    }
    array[i] = threshold;
    QuickSort (array, low, i - 1); // left quickSort
    QuickSort (array, i + 1, high); // right quickSort
}
}
}

```

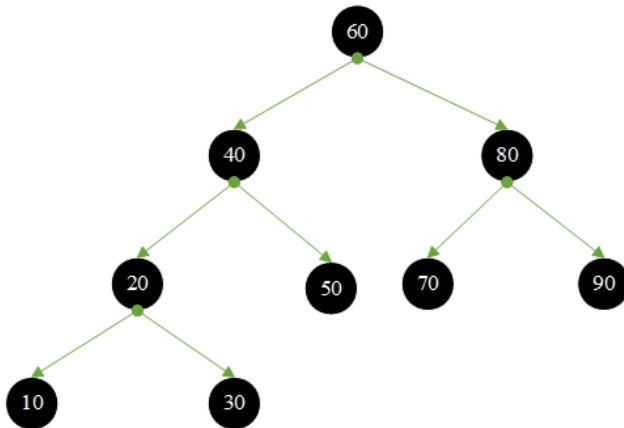
Result:

50,60,70,80,90,100,

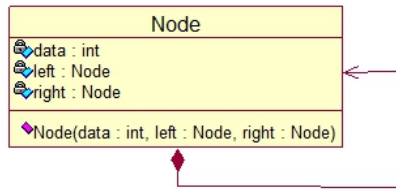
Binary Search Tree

Binary Search Tree:

1. If the left subtree of any node is not empty, the value of all nodes on the left subtree is less than the value of its root node;
2. If the right subtree of any node is not empty, the value of all nodes on the right subtree is greater than the value of its root node;
3. The left subtree and the right subtree of any node are also binary search trees.



Node UML Diagram



Node.cs

```
namespace CSharpDatastructuresAlgorithms
{
    public class Node
    {
        public int data;
        public Node left;
        public Node right;
        public Node(int data, Node left, Node right)
        {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }
}
```

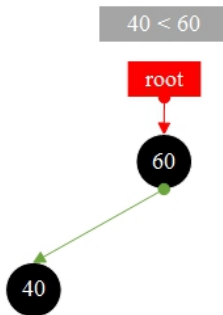
1. Construct a binary search tree, insert node

The inserted nodes are compared from the root node, and the smaller than the root node is compared with the left subtree of the root node, otherwise, compared with the right subtree until the left subtree is empty or the right subtree is empty, then is inserted.

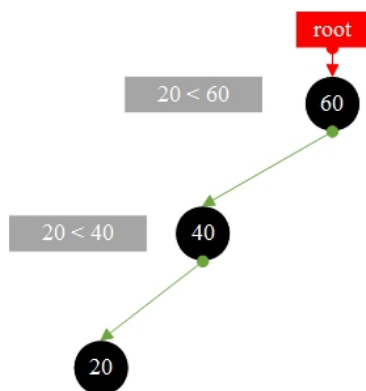
Insert 60



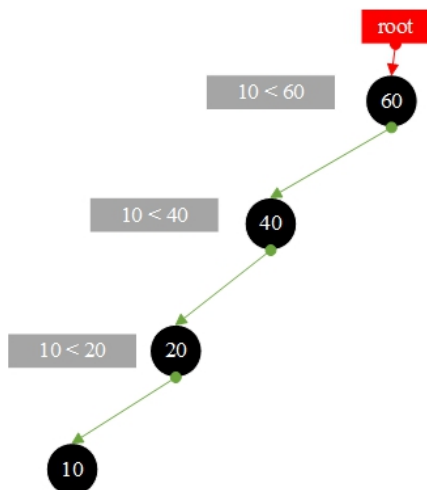
Insert 40



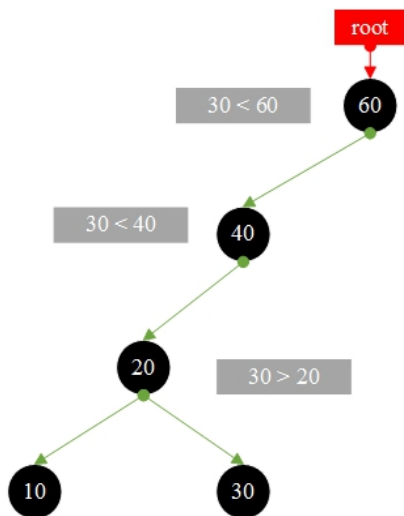
Insert **20**



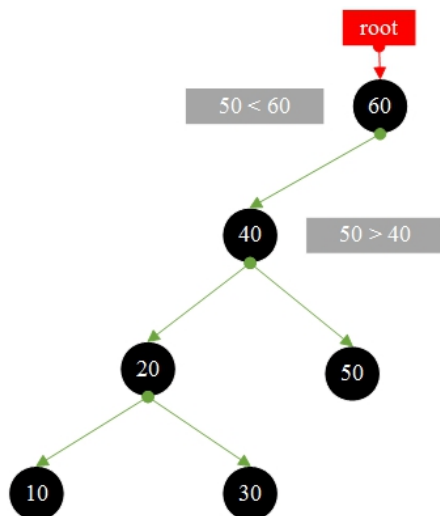
Insert **10**



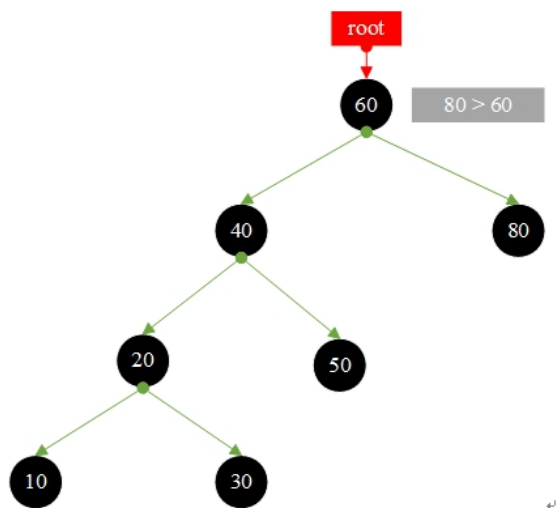
Insert **30**



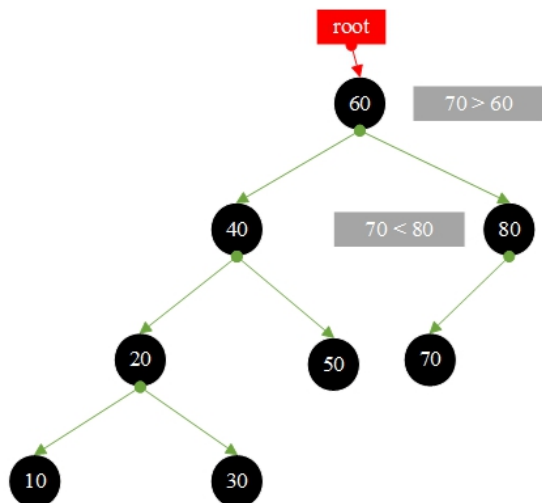
Insert **50**



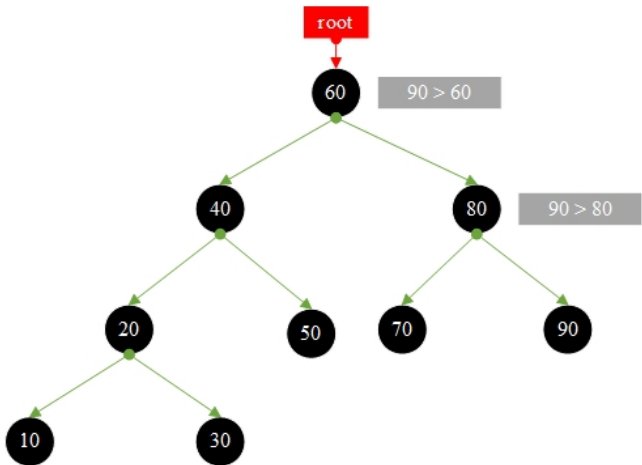
Insert **80**



Insert **70**

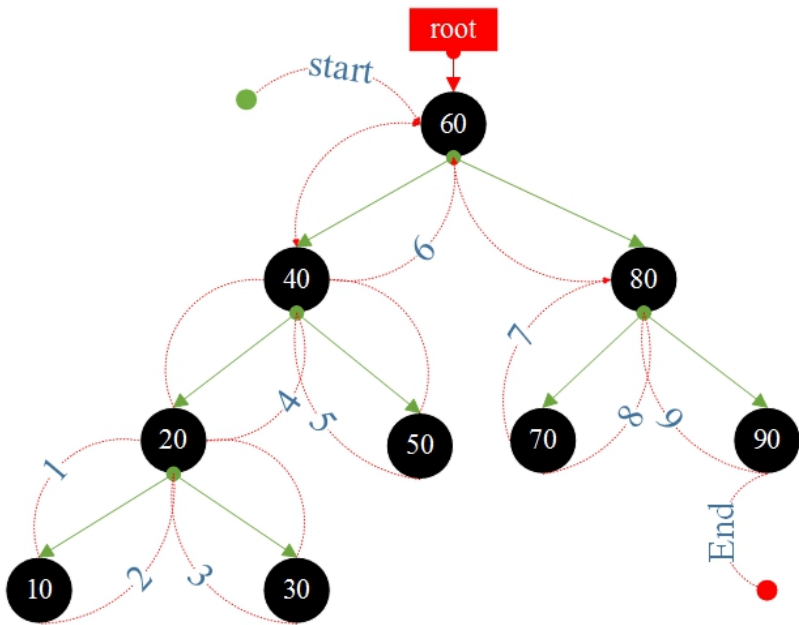


Insert 90

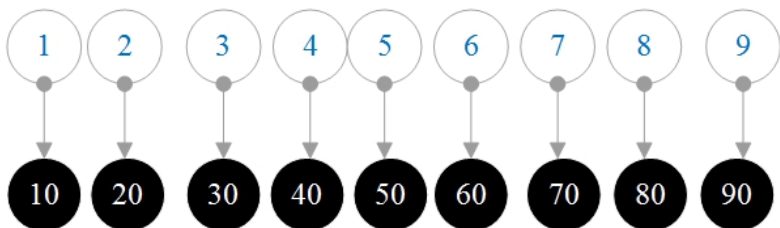


2. binary search tree **In-order traversal**

In-order traversal : left subtree -> root node -> right subtree



Result:



BinaryTree.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class BinaryTree
    {
        private Node root;
        public Node GetRoot()
        {
            return root;
        }
        // In-order traversal binary search tree
        public void InOrder(Node root)
        {
            if (root == null )
            {
                return ;
            }
            InOrder(root.left); // Traversing the left subtree
            Debug.WriteLine(root.data + ", ");
            InOrder(root.right); // Traversing the right subtree
        }
        public void Insert(Node node, int newData)
        {
            if (this.root == null )
            {
                this.root = new Node (newData, null , null );
                return ;
            }
            int compareValue = newData - node.data;
            //Recursive left subtree, continue to find the insertion
            position
            if (compareValue < 0)
            {
                if (node.left == null )
                {
```

```

        node.left = new Node (newData, null , null );
    }
    else
    {
        Insert(node.left, newData);
    }
}
else if (compareValue > 0)
{//////Recursive right subtree, continue to find the
insertion position
    if (node.right == null )
    {
        node.right = new Node (newData, null , null );
    }
    else
    {
        Insert(node.right, newData);
    }
}
}
}
}
}

```

TestBinaryTree.cs

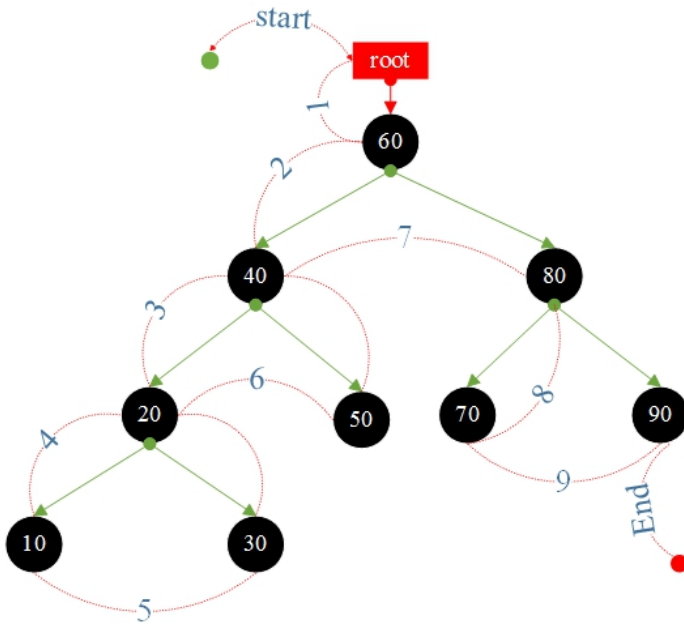
```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestBinaryTree
    {
        public static void Main(String [] args)
        {
            BinaryTree binaryTree = new BinaryTree ();
            //Constructing a binary search tree
            binaryTree.Insert(binaryTree.GetRoot(), 60);
            binaryTree.Insert(binaryTree.GetRoot(), 40);
            binaryTree.Insert(binaryTree.GetRoot(), 20);
            binaryTree.Insert(binaryTree.GetRoot(), 10);
            binaryTree.Insert(binaryTree.GetRoot(), 30);
            binaryTree.Insert(binaryTree.GetRoot(), 50);
            binaryTree.Insert(binaryTree.GetRoot(), 80);
            binaryTree.Insert(binaryTree.GetRoot(), 70);
            binaryTree.Insert(binaryTree.GetRoot(), 90);
            Debug.WriteLine("In-order traversal binary search tree"
);
            binaryTree.InOrder(binaryTree.GetRoot());
        }
    }
}
```

Result:

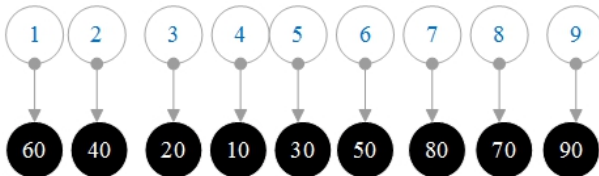
In-order traversal binary search tree
10, 20, 30, 40, 50, 60, 70, 80, 90,

3. binary search tree **Pre-order traversal**

Pre-order traversal : root node -> left subtree -> right subtree



Result:



BinaryTree.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class BinaryTree
    {
        private Node root;
        public Node GetRoot()
        {
            return root;
        }
        //Preorder traversal binary search tree
        public void PreOrder(Node root) {
            if (root == null ) {
                return ;
            }
            Debug .Write(root.data + " , " );
            PreOrder(root.left); // Recursive Traversing the left
            subtree PreOrder(root.right); // Recursive Traversing the right
            subtree
        }
        public void Insert(Node node, int newData)
        {
            if (this .root == null )
            {
                this .root = new Node (newData, null , null );
                return ;
            }
            int compareValue = newData - node.data;
            //Recursive left subtree, continue to find the insertion
            position
            if (compareValue < 0)
            {
                if (node.left == null )
                {
```

```

        node.left = new Node (newData, null , null );
    }
    else
    {
        Insert(node.left, newData);
    }
}
else if (compareValue > 0)
{//////Recursive right subtree, continue to find the
insertion position
    if (node.right == null )
    {
        node.right = new Node (newData, null , null );
    }
    else
    {
        Insert(node.right, newData);
    }
}
}
}
}
}

```

TestBinaryTree.cs

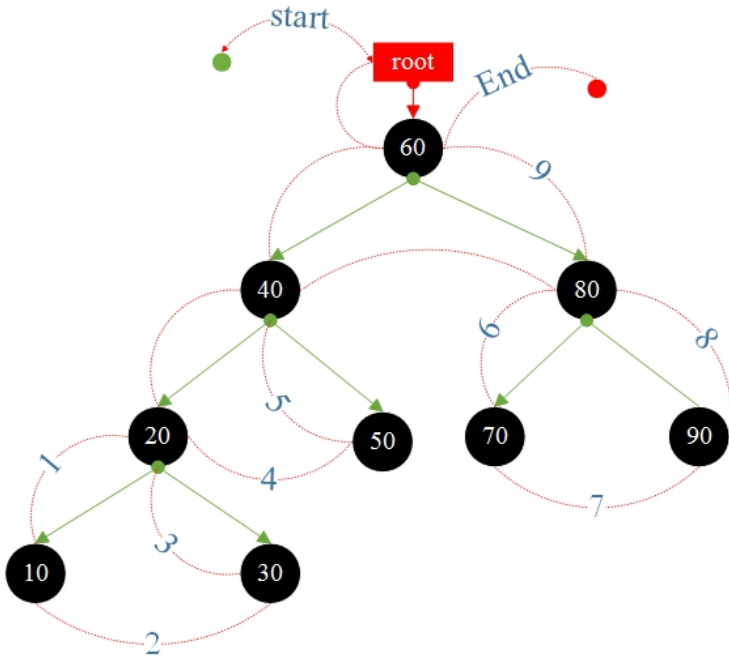
```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestBinaryTree
    {
        public static void Main(String [] args)
        {
            BinaryTree binaryTree = new BinaryTree ();
            //Constructing a binary search tree
            binaryTree.Insert(binaryTree.GetRoot(), 60);
            binaryTree.Insert(binaryTree.GetRoot(), 40);
            binaryTree.Insert(binaryTree.GetRoot(), 20);
            binaryTree.Insert(binaryTree.GetRoot(), 10);
            binaryTree.Insert(binaryTree.GetRoot(), 30);
            binaryTree.Insert(binaryTree.GetRoot(), 50);
            binaryTree.Insert(binaryTree.GetRoot(), 80);
            binaryTree.Insert(binaryTree.GetRoot(), 70);
            binaryTree.Insert(binaryTree.GetRoot(), 90);
            Debug.WriteLine("Pre-order traversal binary search tree"
);
            binaryTree.PreOrder(binaryTree.GetRoot());
        }
    }
}
```

Result:

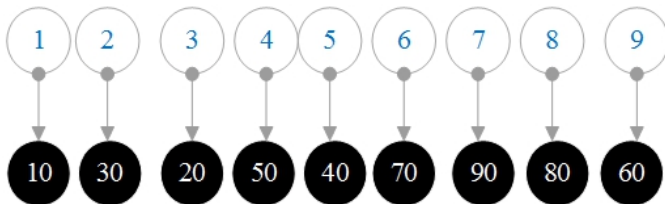
Pre-order traversal binary search tree
60, 40, 20, 10, 30, 50, 80, 70, 90,

4. binary search tree **Post-order traversal**

Post-order traversal : right subtree -> root node -> left subtree



Result:



BinaryTree.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class BinaryTree
    {
        private Node root;
        public Node GetRoot()
        {
            return root;
        }
        //Post-order traversal binary search tree
        public void PostOrder(Node root)
        {
            if (root == null )
            {
                return ;
            }
            PostOrder(root.left); // Recursive Traversing the left
            subtree PostOrder(root.right); // Recursive Traversing the right
            subtree Debug .Write(root.data + ", " );
        }
        public void Insert(Node node, int newData)
        {
            if (this .root == null )
            {
                this .root = new Node (newData, null , null );
                return ;
            }
            int compareValue = newData - node.data;
            //Recursive left subtree, continue to find the insertion
            position if (compareValue < 0)
            {

```

```

        if (node.left == null )
        {
            node.left = new Node (newData, null , null );
        }
        else
        {
            Insert(node.left, newData);
        }
    }
    else if (compareValue > 0)
    {////Recursive right subtree, continue to find the
insertion position
        if (node.right == null )
        {
            node.right = new Node (newData, null , null );
        }
        else
        {
            Insert(node.right, newData);
        }
    }
}
}
}
}

```

TestBinaryTree.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestBinaryTree
    {
        public static void Main(String [] args)
        {
            BinaryTree binaryTree = new BinaryTree ();
            //Constructing a binary search tree
            binaryTree.Insert(binaryTree.GetRoot(), 60);
            binaryTree.Insert(binaryTree.GetRoot(), 40);
            binaryTree.Insert(binaryTree.GetRoot(), 20);
            binaryTree.Insert(binaryTree.GetRoot(), 10);
            binaryTree.Insert(binaryTree.GetRoot(), 30);
            binaryTree.Insert(binaryTree.GetRoot(), 50);
            binaryTree.Insert(binaryTree.GetRoot(), 80);
            binaryTree.Insert(binaryTree.GetRoot(), 70);
            binaryTree.Insert(binaryTree.GetRoot(), 90);
            Debug.WriteLine("Post-order traversal binary search
tree" );
            binaryTree.PostOrder(binaryTree.GetRoot());
        }
    }
}
```

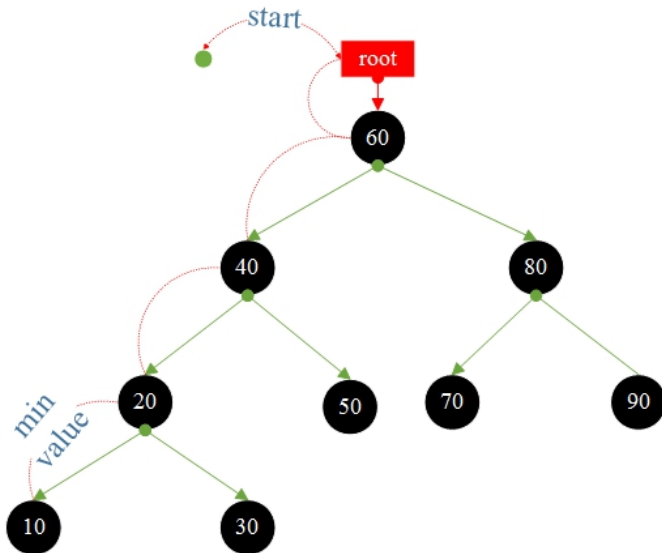
Result:

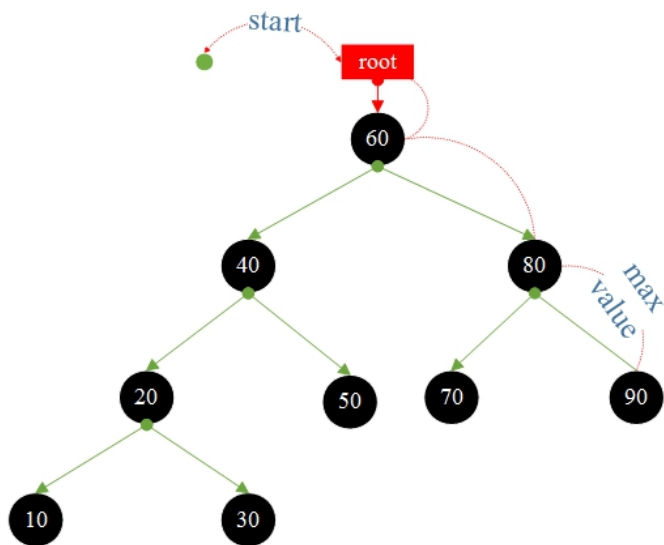
Post-order traversal binary search tree
10, 30, 20, 50, 40, 70, 90, 80, 60,

5. binary search tree **Maximum and minimum**

Minimum value: The small value is on the left child node, as long as the recursion traverses the left child until be empty, the current node is the minimum node.

Maximum value: The large value is on the right child node, as long as the recursive traversal is the right child until be empty, the current node is the largest node.





BinaryTree.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class BinaryTree
    {
        private Node root;
        public Node GetRoot()
        {
            return root;
        }
        //Minimum value
        public Node SearchMinValue(Node node)
        {
            if (node == null || node.data == 0)
                return null ;
            if (node.left == null )
            {
                return node;
            }
            return SearchMinValue(node.left); //Recursively find
            minimum from the left subtree
        }
        //Maximum value
        public Node SearchMaxValue(Node node)
        {
            if (node == null || node.data == 0)
                return null ;
            if (node.right == null )
            {
                return node;
            }
            return SearchMaxValue(node.right); //Recursively find
            minimum from the right subtree
        }
        public void Insert(Node node, int newData)
```

```

{
    if (this .root == null )
    {
        this .root = new Node (newData, null , null );
        return ;
    }
    int compareValue = newData - node.data;
    //Recursive left subtree, continue to find the insertion
position
    if (compareValue < 0)
    {
        if (node.left == null )
        {
            node.left = new Node (newData, null , null );
        }
        else
        {
            Insert(node.left, newData);
        }
    }
    else if (compareValue > 0)
    {
        //Recursive right subtree, continue to find the
insertion position
        if (node.right == null )
        {
            node.right = new Node (newData, null , null );
        }
        else
        {
            Insert(node.right, newData);
        }
    }
}
}
}

```


TestBinaryTree.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestBinaryTree
    {
        public static void Main(String [] args)
        {
            BinaryTree binaryTree = new BinaryTree ();
            //Constructing a binary search tree
            binaryTree.Insert(binaryTree.GetRoot(), 60);
            binaryTree.Insert(binaryTree.GetRoot(), 40);
            binaryTree.Insert(binaryTree.GetRoot(), 20);
            binaryTree.Insert(binaryTree.GetRoot(), 10);
            binaryTree.Insert(binaryTree.GetRoot(), 30);
            binaryTree.Insert(binaryTree.GetRoot(), 50);
            binaryTree.Insert(binaryTree.GetRoot(), 80);
            binaryTree.Insert(binaryTree.GetRoot(), 70);
            binaryTree.Insert(binaryTree.GetRoot(), 90);
            Debug.WriteLine("\nMinimum Value" );
            Node
minNode = binaryTree.SearchMinValue(binaryTree.GetRoot());
            Debug.WriteLine(minNode.data);
            Debug.WriteLine("\nMaximum Value" );
            Node
maxNode = binaryTree.SearchMaxValue(binaryTree.GetRoot());
            Debug.WriteLine(maxNode.data);
        }
    }
}
```

Result:

Minimum Value

10

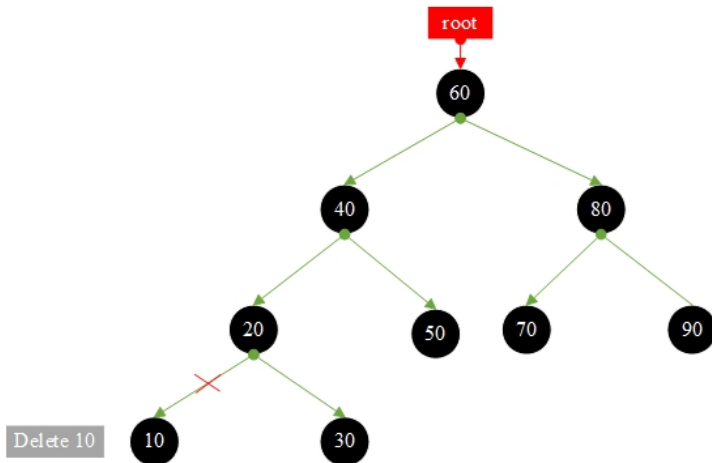
Maximum Value

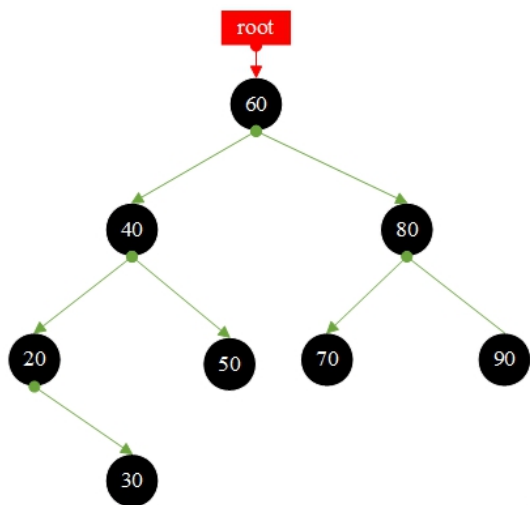
6. binary search tree **Delete Node**

Binary search tree delete node 3 cases

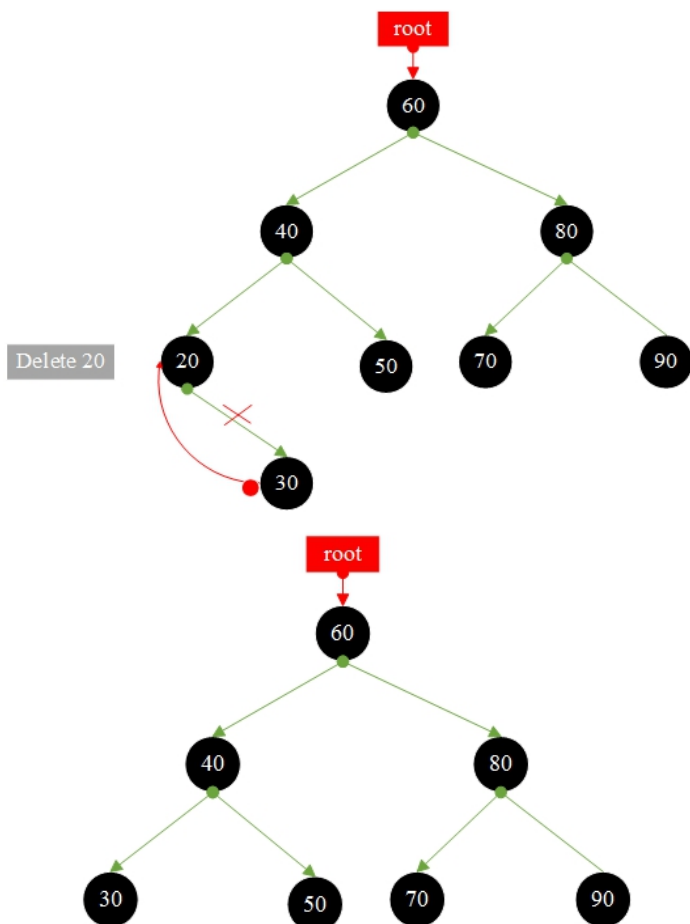
1. If there is no child node, delete it directly
2. If there is only one child node, the child node replaces the current node, and then deletes the current node.
3. If there are two child nodes, replace the current node with the smallest node from the right subtree, because the smallest node on the right is also larger than the value on the left.

1. If there is no child node, delete it directly: **delete node 10**

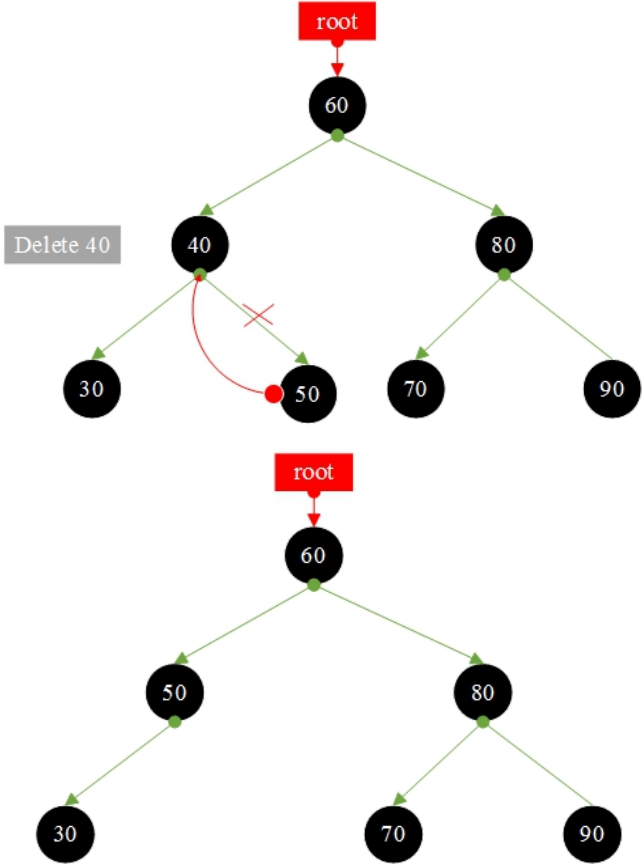




2. If there is only one child node, the child node replaces the current node, and then deletes the current node. **Delete node 20**



3. If there are two child nodes, replace the current node with the smallest node from the right subtree, **Delete node 40**



BinaryTree.cs

TestBinaryTree.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestBinaryTree
    {
        public static void Main(String [] args)
        {
            BinaryTree binaryTree = new BinaryTree ();
            binaryTree.Insert(binaryTree.GetRoot(), 60);
            binaryTree.Insert(binaryTree.GetRoot(), 40);
            binaryTree.Insert(binaryTree.GetRoot(), 20);
            binaryTree.Insert(binaryTree.GetRoot(), 10);
            binaryTree.Insert(binaryTree.GetRoot(), 30);
            binaryTree.Insert(binaryTree.GetRoot(), 50);
            binaryTree.Insert(binaryTree.GetRoot(), 80);
            binaryTree.Insert(binaryTree.GetRoot(), 70);
            binaryTree.Insert(binaryTree.GetRoot(), 90);
            Debug.WriteLine("\ndelete node is: 10" );
            binaryTree.Remove(binaryTree.GetRoot(), 10);
            Debug.WriteLine("\nIn-order traversal binary tree" );
            binaryTree.InOrder(binaryTree.GetRoot());
            Debug.WriteLine("\n-----" );
            Debug.WriteLine("\ndelete node is: 20" );
            binaryTree.Remove(binaryTree.GetRoot(), 20);
            Debug.WriteLine("\nIn-order traversal binary tree" );
            binaryTree.InOrder(binaryTree.GetRoot());
            Debug.WriteLine("\n-----" );
            Debug.WriteLine("\ndelete node is: 40" );
            binaryTree.Remove(binaryTree.GetRoot(), 40);
            Debug.WriteLine("\nIn-order traversal binary tree" );
            binaryTree.InOrder(binaryTree.GetRoot());
        }
    }
}
```


Result:

delete node is: 10

In-order traversal binary tree
20, 30, 40, 50, 60, 70, 80, 90,

delete node is: 20

In-order traversal binary tree
30, 40, 50, 60, 70, 80, 90,

delete node is: 40

In-order traversal binary tree
30, 50, 60, 70, 80, 90,

Binary Heap Sorting

Binary Heap Sorting:

The value of the non-terminal node in the binary tree is not greater than the value of its left and right child nodes.

Small top heap : $k_i \leq k_{2i}$ and $k_i \leq k_{2i+1}$

Big top heap : $k_i \geq k_{2i}$ and $k_i \geq k_{2i+1}$

Parent node subscript = $(i-1)/2$

Left subnode subscript = $2*i + 1$

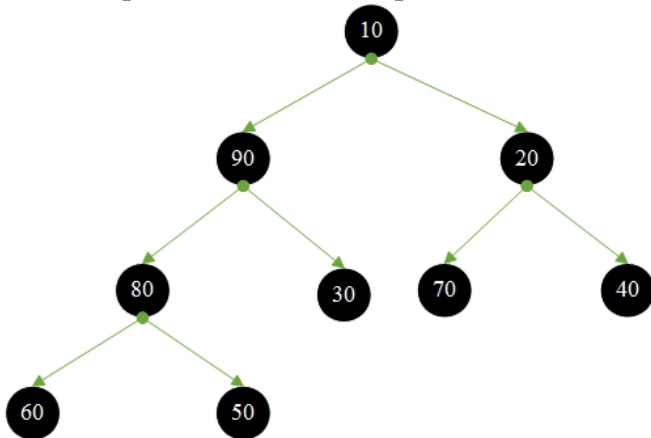
Right subnode subscript = $2*i + 2$

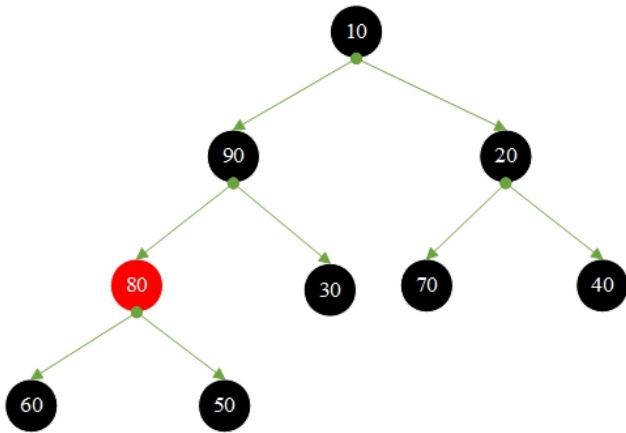
Heap sorting process:

1. Build a heap
2. After outputting the top element of the heap, adjust from top to bottom, compare the top element with the root node of its left and right subtrees, and swap the smallest element to the top of the heap; then adjust continuously until the leaf nodes to get new heap.

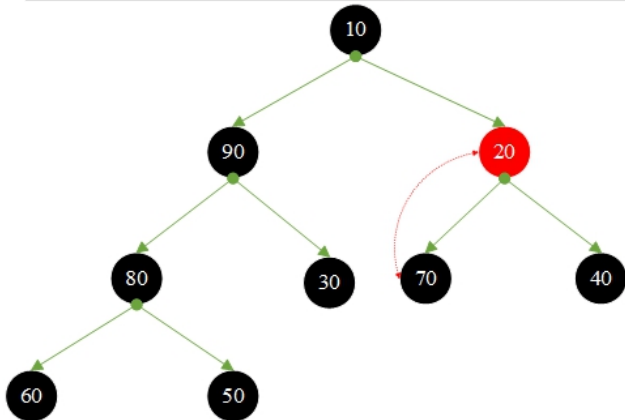
1. **{10, 90, 20, 80, 30, 70, 40, 60, 50}** build heap and then heap sort output.

Initialize the heap and build the heap

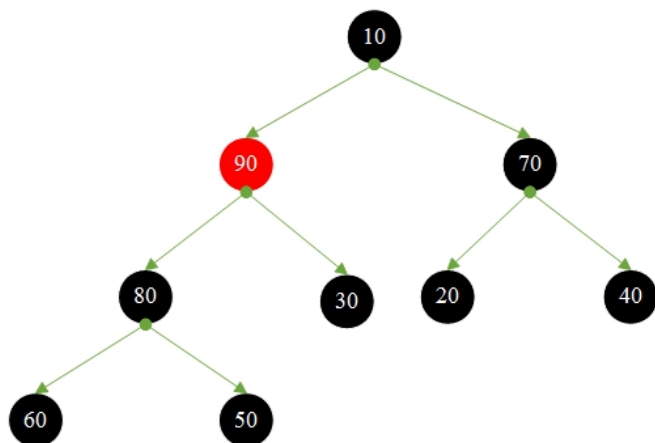




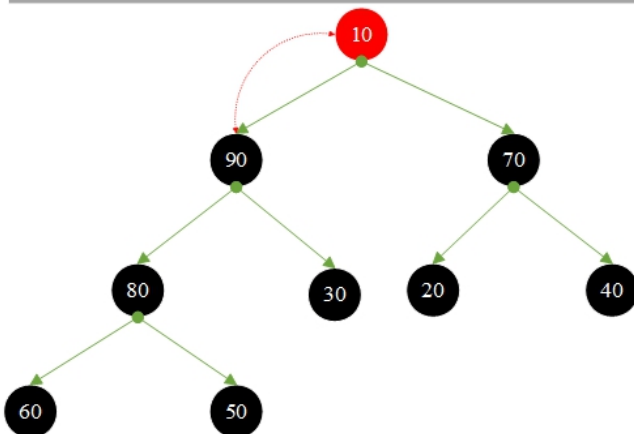
Not Leaf Node = 80 > left = 60 , 80 > right = 50 No need to move



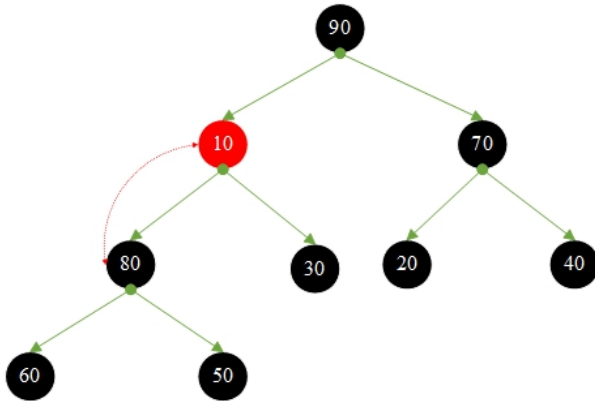
Not Leaf Node = 20 < left = 70 , 70 > right = 40 , 20 swap with 70



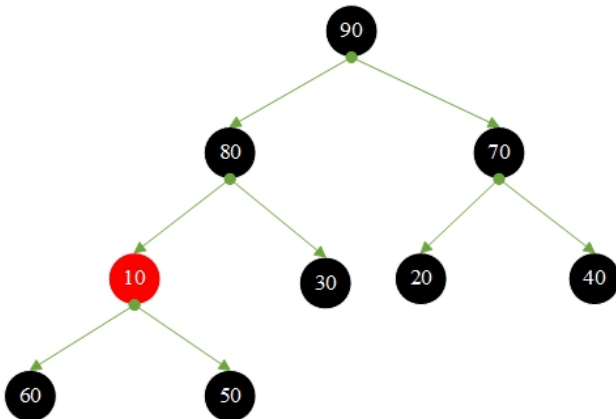
Not Leaf Node = $90 > \text{left} = 80$, $80 > \text{right} = 30$ No need to move



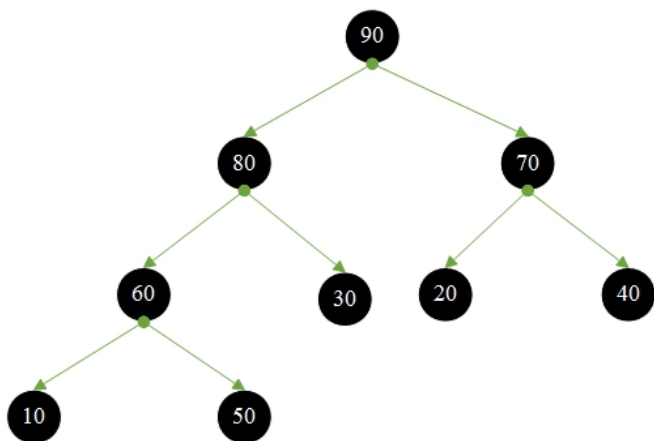
Not Leaf Node = $10 < \text{left} = 90$, $90 > \text{right} = 70$, 10 swap with 90



Still Not Leaf Node = 10 < left = 80 , 80 > right = 30 , 10 swap with 80

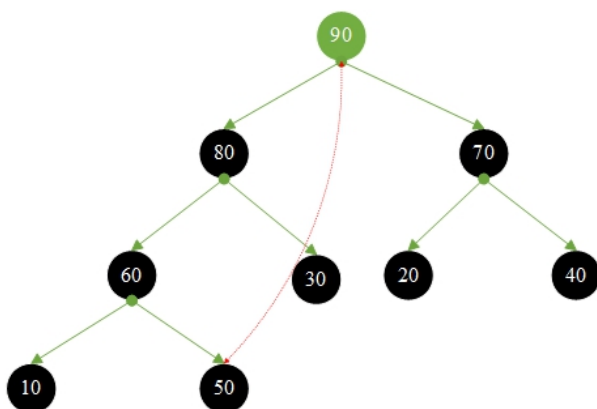


Still Not Leaf Node = 10 < left = 60 , 60 > right = 50 , 10 swap with 60

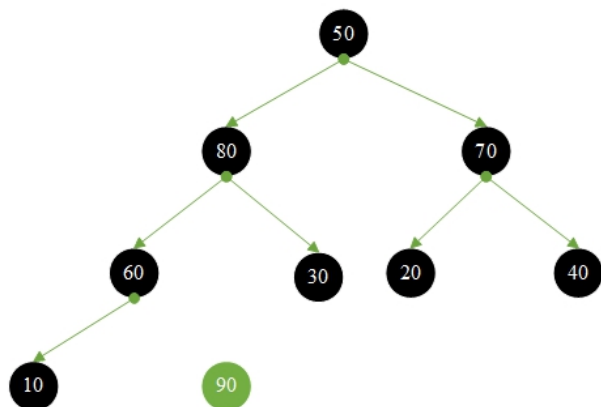


Create the heap finished

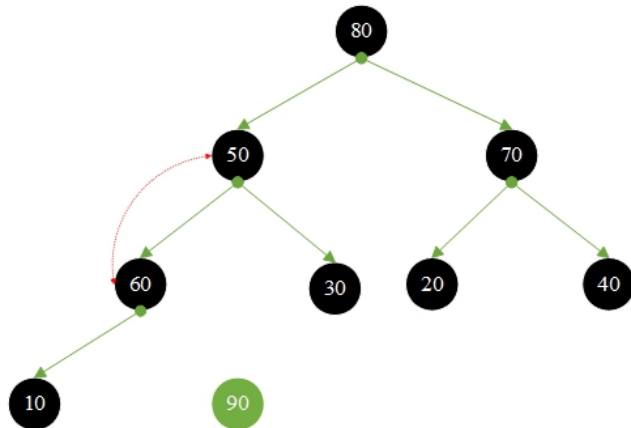
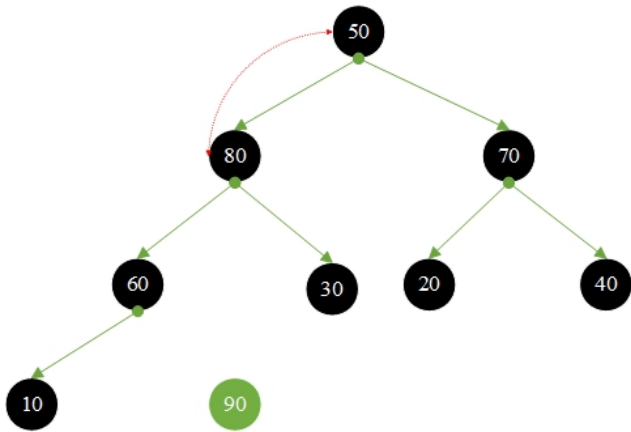
2. Start heap sorting

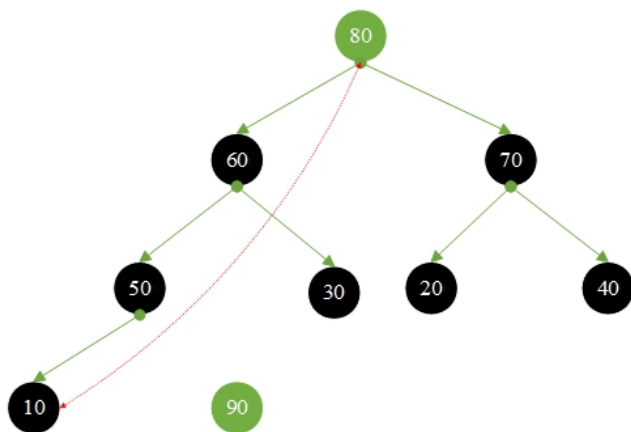


root = 90 and tail = 50 are exchanged

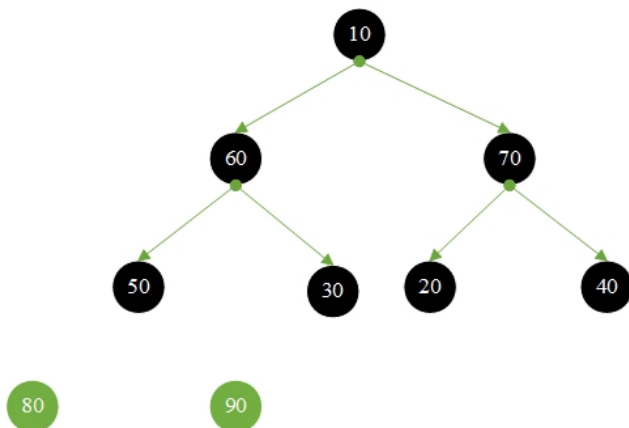


adjust the heap

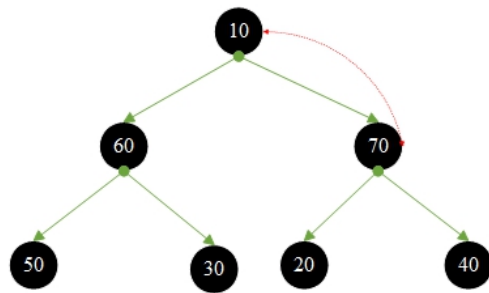




root = 80 and tail = 10 are exchanged

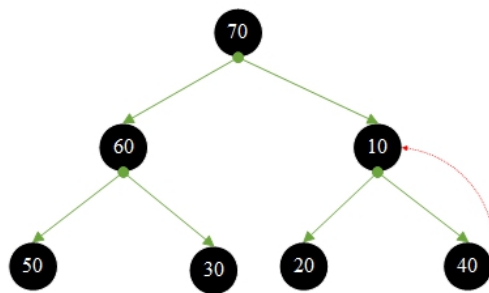


adjust the heap



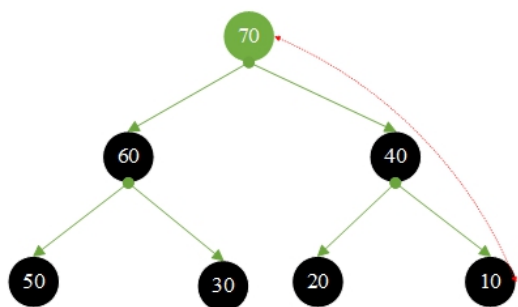
80

90



80

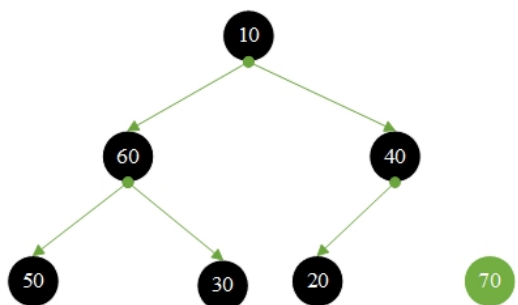
90



80

90

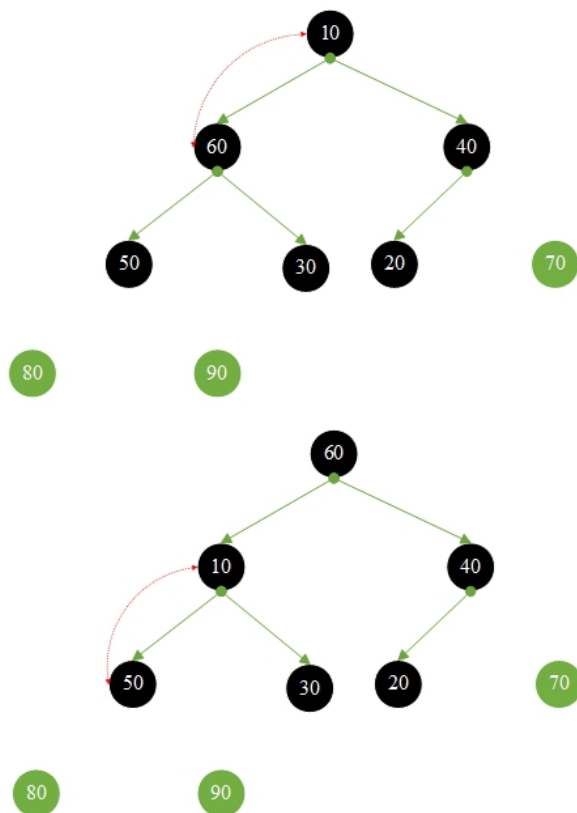
root = 70 and tail = 10 are exchanged

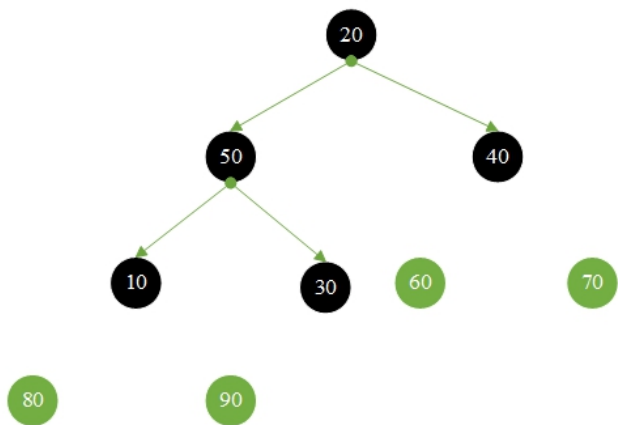
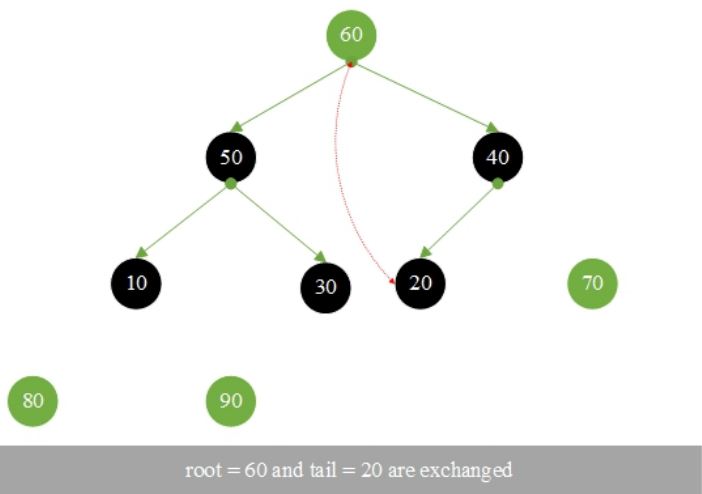


80

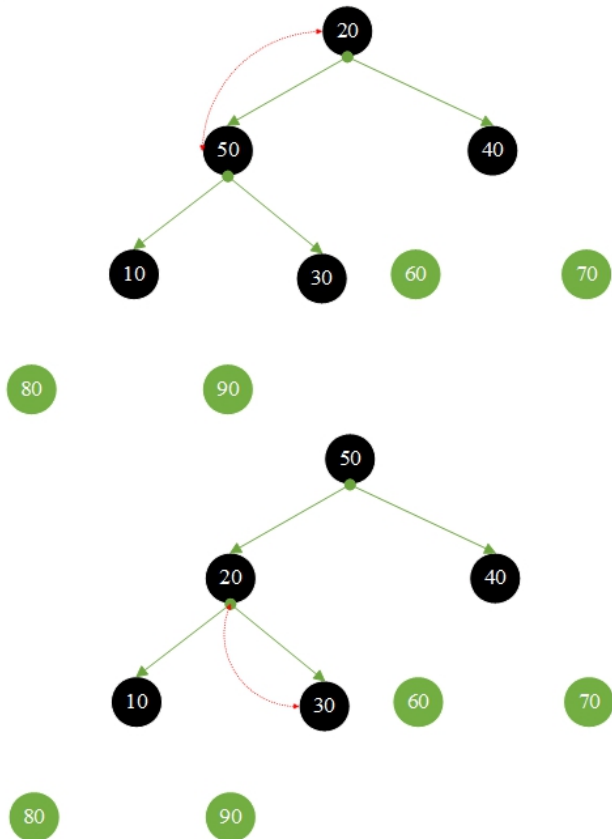
90

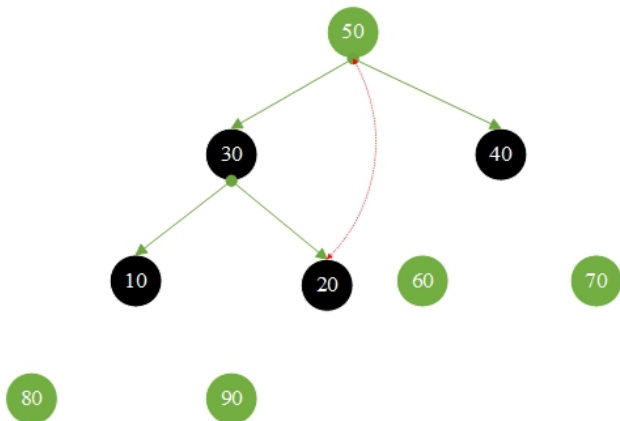
adjust the heap



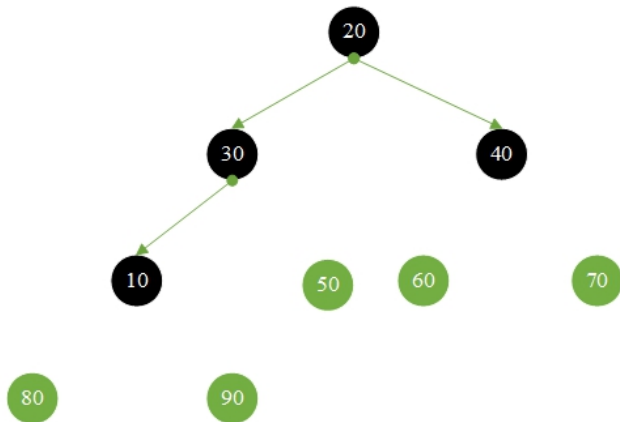


adjust the heap

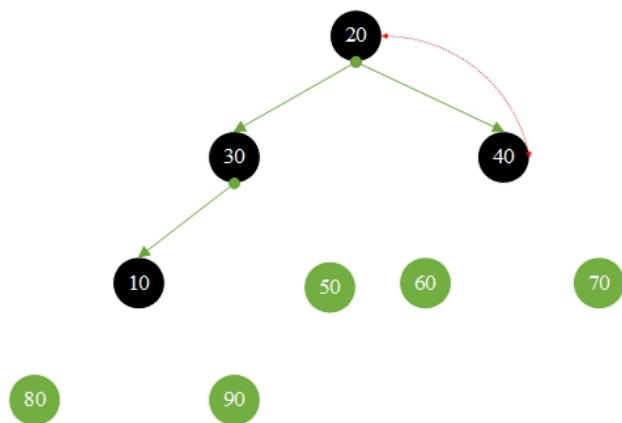


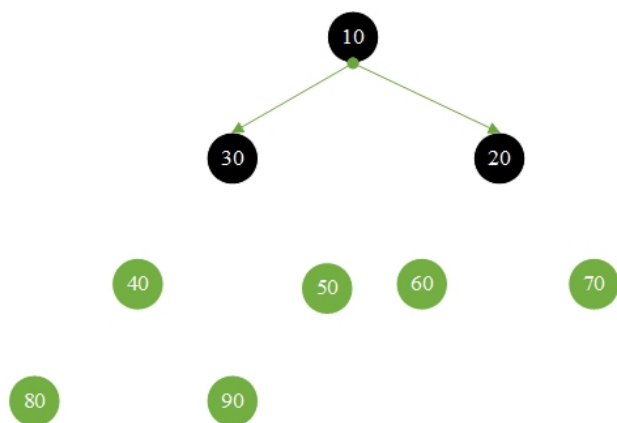
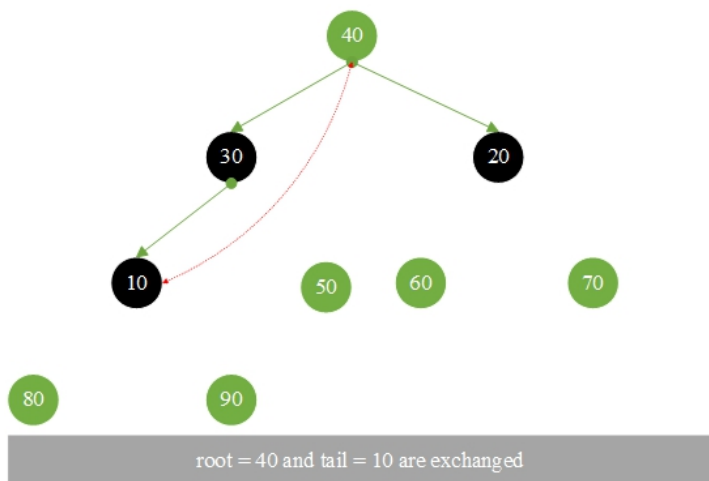


root = 50 and tail = 20 are exchanged

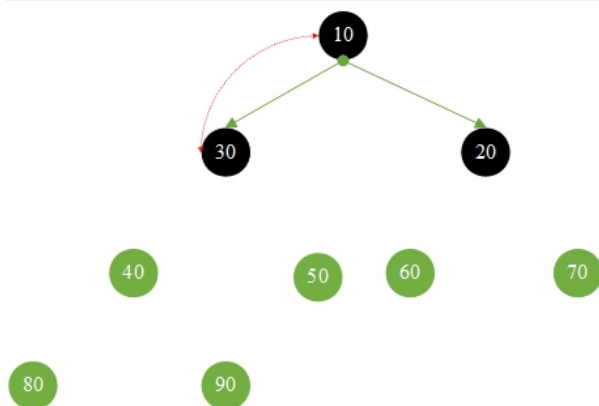


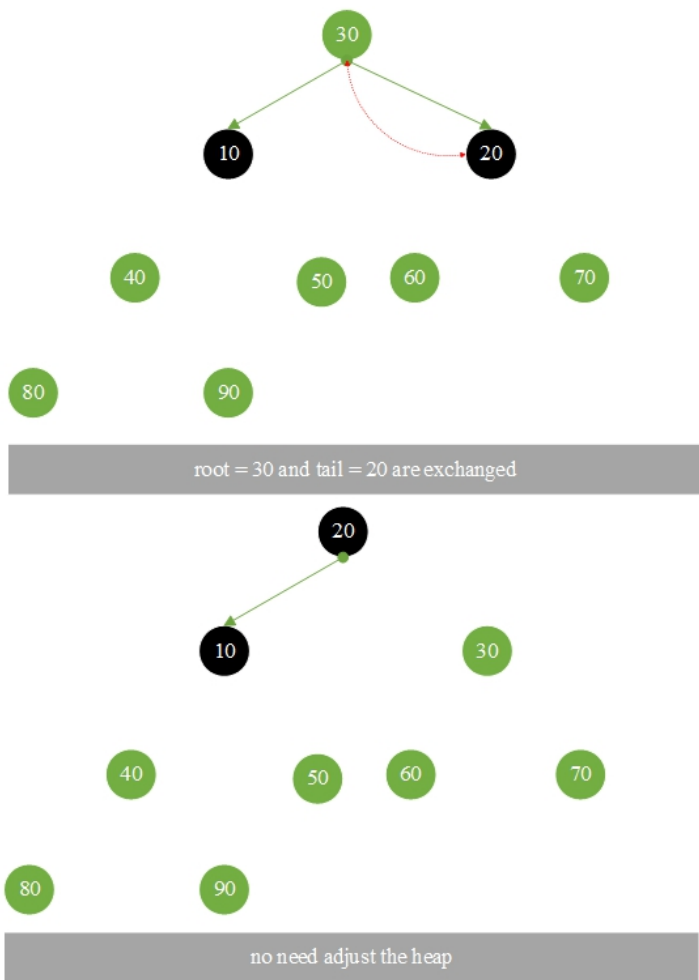
adjust the heap

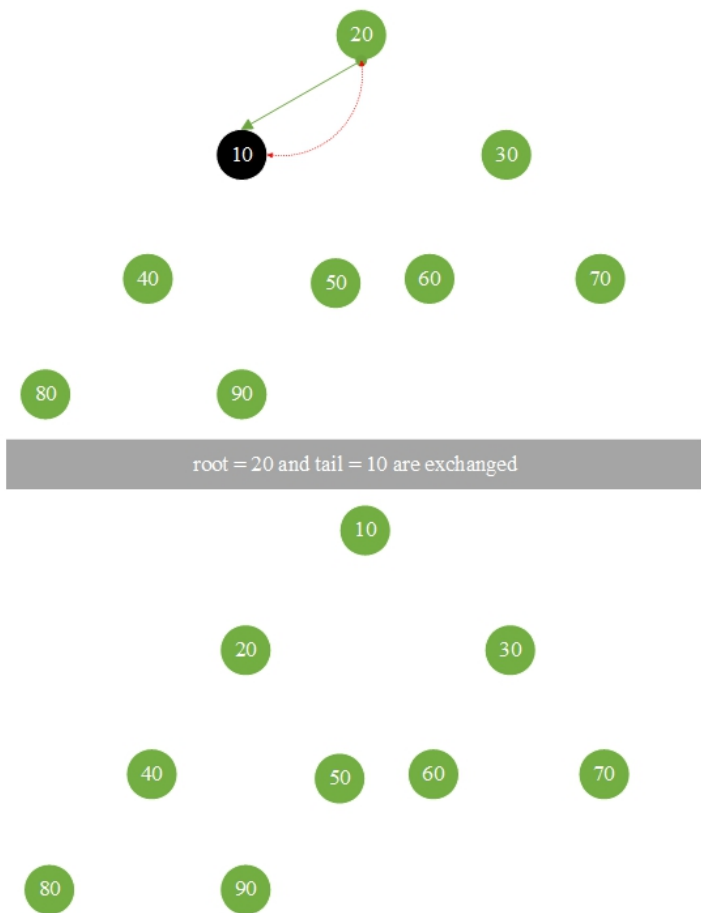




adjust the heap







Heap sort result



HeapSort.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class HeapSort
    {
        private int [] array;
        //Initialize the heap
        public void CreateHeap(int [] array)
        {
            this.array = array;
            // Build a heap, (array.Length - 1) / 2 scan half of the
nodes with child nodes
            for (int i = (array.Length - 1) / 2; i >= 0; i--)
            {
                AdjustHeap(i, array.Length - 1);
            }
        }
        //Adjustment heap
        public void AdjustHeap(int currentIndex, int maxLength)
        {
            int noLeafValue = array[currentIndex]; // Current non-
leaf node
            //2 * currentIndex + 1 Current left subtree subscript
            for (int j = 2 * currentIndex + 1; j <= maxLength; j =
currentIndex * 2 + 1)
            {
                if (j < maxLength && array[j] < array[j + 1])
                {
                    j++; // j Large subscript
                }
                if (noLeafValue >= array[j])
```

```

        {
            break ;
        }
        array[currentIndex] = array[j]; // Move up to the
parent node
        currentIndex = j;
    }
    array[currentIndex] = noLeafValue; // To put in the
position
}
public void HeapSort()
{
    for (int i = array.Length - 1; i > 0; i--)
    {
        int temp = array[0];
        array[0] = array[i];
        array[i] = temp;
        AdjustHeap(0, i - 1);
    }
}
}
}

```

TestHeapSort.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestHeapSort
    {
        public static void Main(String [] args)
        {
            HeapSort heapSort = new HeapSort ();
            int [] scores = { 10, 90, 20, 80, 30, 70, 40, 60, 50 };
            Debug.WriteLine("Before building a heap : ");
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.Write(scores[i] + ", ");
            }
            Debug.WriteLine("\n\n");
            //////////////////////////////////////
            Debug.WriteLine("After building a heap : ");
            heapSort.CreateHeap(scores);
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.Write(scores[i] + ", ");
            }
            Debug.WriteLine("\n\n");
            //////////////////////////////////////
            Debug.WriteLine("After heap sorting : ");
            heapSort.HeapSort();
            for (int i = 0; i < scores.Length; i++)
            {
                Debug.Write(scores[i] + ", ");
            }
        }
    }
}
```

Result:

Before building a heap :

10, 90, 20, 80, 30, 70, 40, 60, 50,

After building a heap :

90, 80, 70, 60, 30, 20, 40, 10, 50,

After heap sorting :

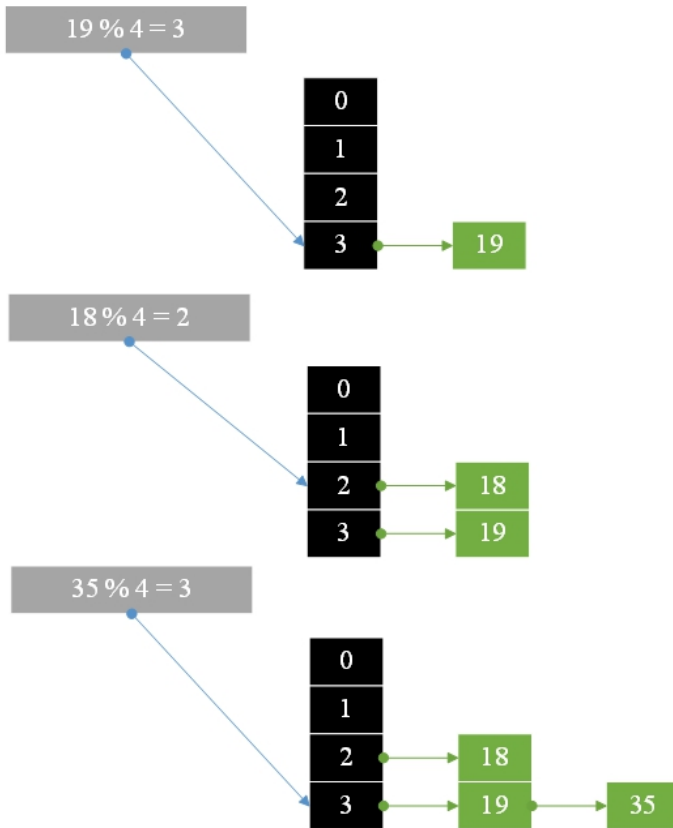
10, 20, 30, 40, 50, 60, 70, 80, 90,

Hash Table

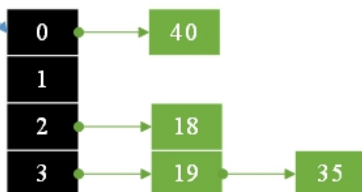
Hash Table:

Access by mapping key $\Rightarrow$ values in the table.

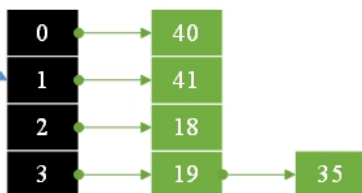
1. Map {19, 18, 35, 40, 41, 42} to the HashTable mapping rule $\text{key \% 4}$



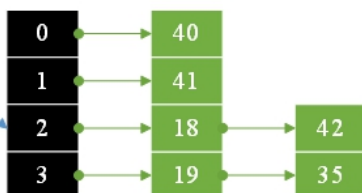
$$40 \% 4 = 0$$



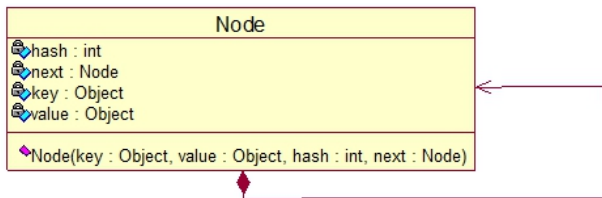
$$41 \% 4 = 1$$



$$42 \% 4 = 2$$



2. Implement a Hashtable



Node.cs

```
namespace CSharpDatastructuresAlgorithms
{
    public class Node
    {
        public Object key;
        public Object value;
        public int hash;
        public Node next;
        public Node(Object key, Object value, int hash, Node next)
        {
            this.key = key;
            this.value = value;
            this.hash = hash;
            this.next = next;
        }
    }
}
```

Hashtable.cs

TestHashtable.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    class TestHashtable
    {
        public static void Main(String [] args)
        {
            Hashtable table = new Hashtable ();
            table.Put("david" , "Good Boy Keep Going" );
            table.Put("grace" , "Cute Girl Keep Going" );
            Debug.WriteLine("david => " + table.Get("david" ));
            Debug.WriteLine("grace => " + table.Get("grace" ));
        }
    }
}
```

Result:

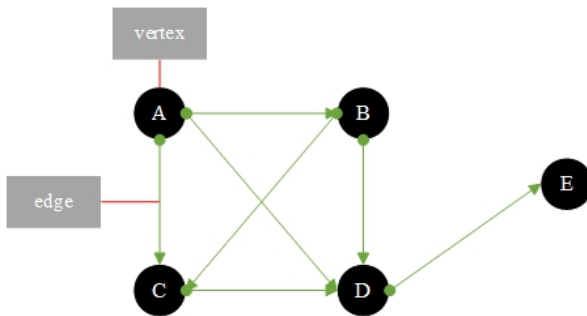
```
david => Good Boy Keep Going
grace => Cute Girl Keep Going
```

Directed Graph and Depth-First Search

Directed Graph:

The data structure is represented by an adjacency matrix (that is, a two-dimensional array) and an adjacency list. Each node is called a vertex, and two adjacent nodes are called edges.

Directed Graph has direction : **A -> B** and **B -> A** are different



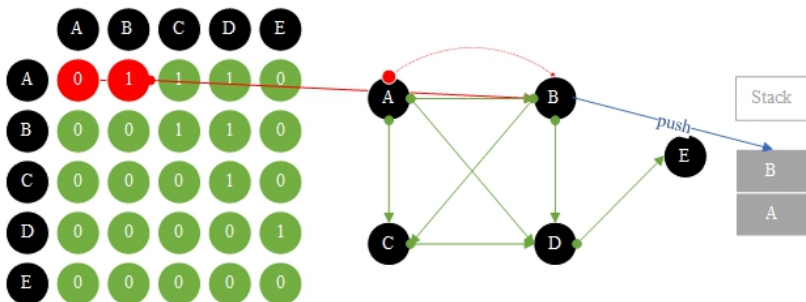
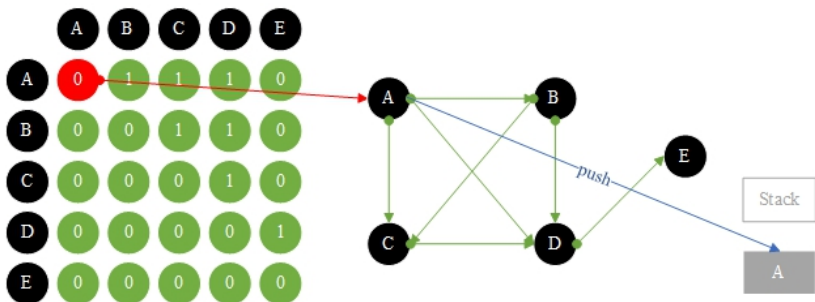
1. The adjacency matrix is described above:

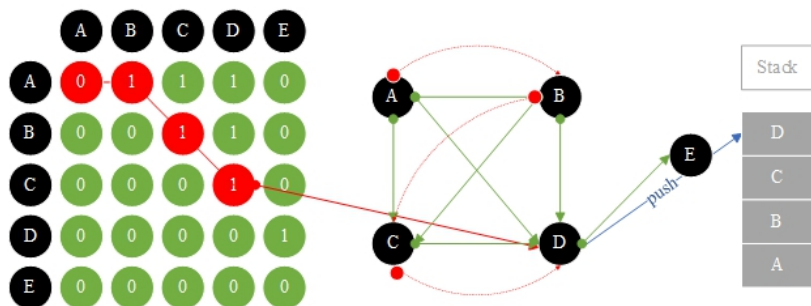
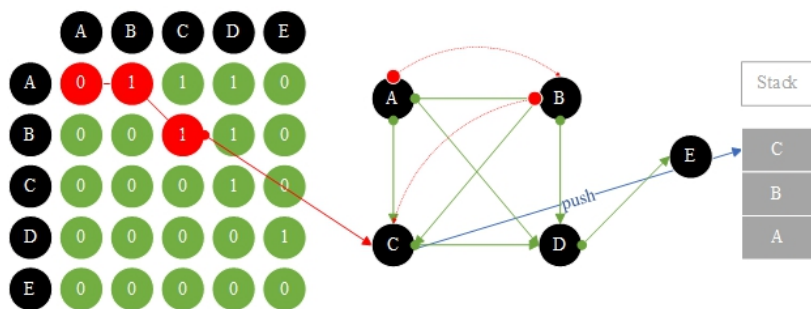
The total number of vertices is a two-dimensional array size, if have value of the edge is **1** , otherwise no value of the edge is **0** .

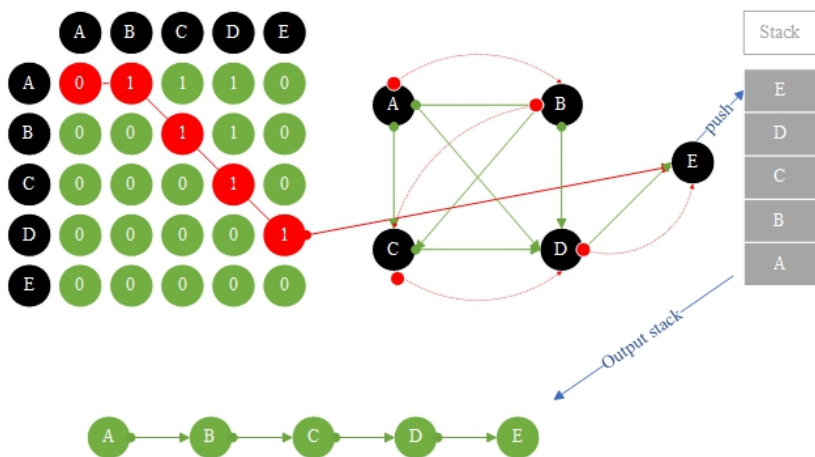
	A	B	C	D	E
A	0	1	1	1	0
B	0	0	1	1	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

2. Depth-First Search:

Look for the neighboring edge node B from A and then find the neighboring node C from B and so on until all nodes are found **A -> B -> C -> D -> E**.







Vertex.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    public class Vertex
    {
        public String data;
        public bool visited; // Have you visited
        public Vertex(String data, bool visited)
        {
            this.data = data;
            this.visited = visited;
        }
    }
}
```

Graph.cs

TestGraph.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    public class TestGraph
    {
        public static void Main(String [] args)
        {
            Graph graph = new Graph (5);
            graph.AddVertex("A" );
            graph.AddVertex("B" );
            graph.AddVertex("C" );
            graph.AddVertex("D" );
            graph.AddVertex("E" );
            graph.AddEdge(0, 1);
            graph.AddEdge(0, 2);
            graph.AddEdge(0, 3);
            graph.AddEdge(1, 2);
            graph.AddEdge(1, 3);
            graph.AddEdge(2, 3);
            graph.AddEdge(3, 4);
            // Two-dimensional array traversal output vertex edge
            and adjacent array
            PrintGraph(graph);
            Debug .Write("\nDepth-first search traversal output : \n"
);
            graph.DepthFirstSearch();
        }
        public static void PrintGraph(Graph graph)
        {
            Debug .Write("Two-dimensional array traversal vertex
            edge and adjacent array : \n  ");
            for (int i = 0; i < graph.GetVertexs().Length; i + + )
            {
                Debug .Write(graph.GetVertexs()[i].data + "  ");
            }
        }
    }
}
```

```

        Debug.WriteLine("\n");
        for (int i = 0; i <
graph.GetAdjacencyMatrix().GetLength(0); i++)
        {
            Debug.WriteLine(graph.GetVertexs()[i].data + " ");
            for (int j = 0; j <
graph.GetAdjacencyMatrix().GetLength(0); j++)
            {
                Debug.WriteLine(graph.GetAdjacencyMatrix()[i, j] +
" ");
            }
            Debug.WriteLine("\n");
        }
    }
}

```

Result:

	A	B	C	D	E
A	0	1	1	1	0
B	0	0	1	1	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

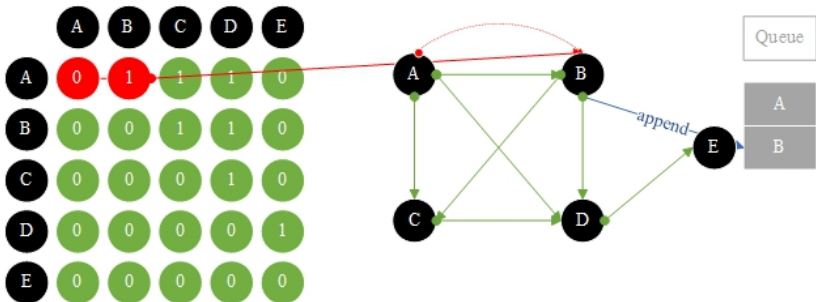
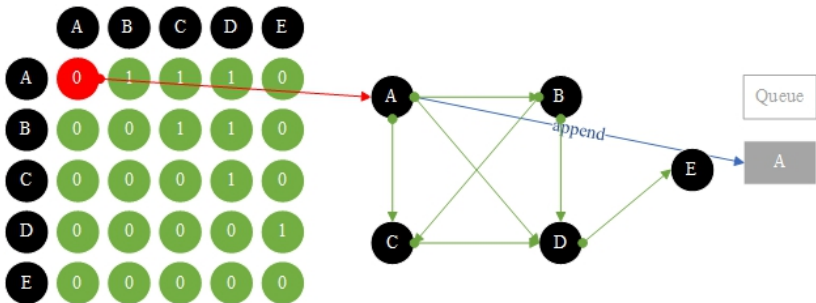
Depth-first search traversal output :

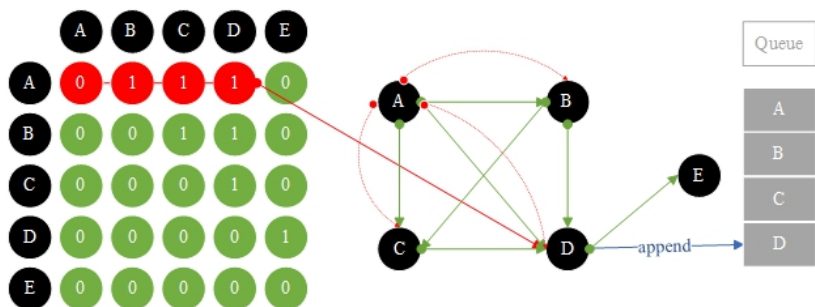
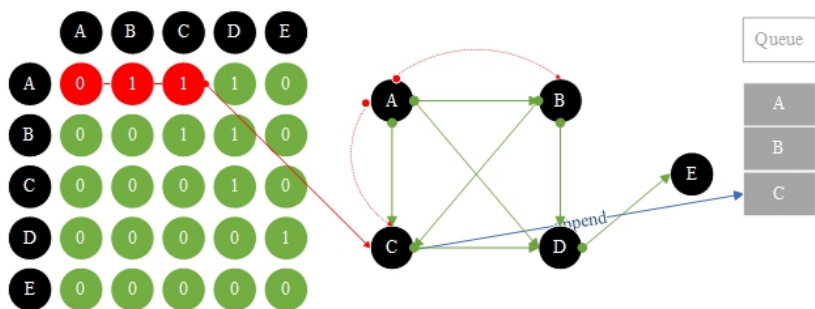
A -> B -> C -> D -> E

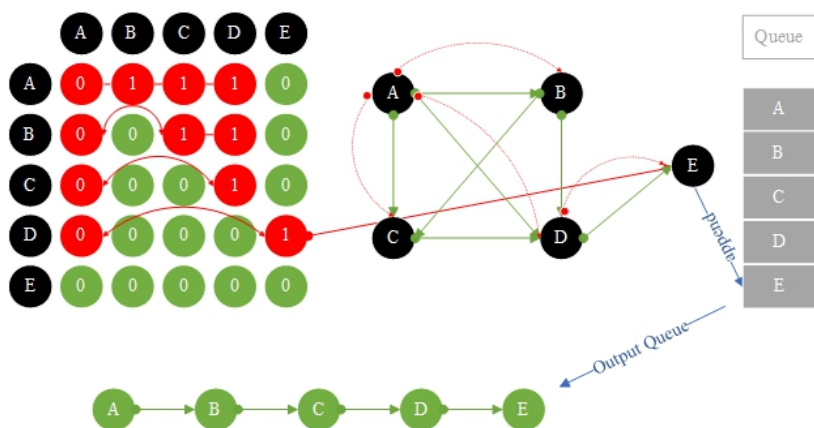
Directed Graph and Breadth-First Search

Breadth-First Search:

Find all neighboring edge nodes B, C, D from A and then find all neighboring nodes A, C, D from B and so on until all nodes are found **A -> B -> C -> D -> E** .







Vertex.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    public class Vertex
    {
        public String data;
        public bool visited; // Have you visited
        public Vertex(String data, bool visited)
        {
            this.data = data;
            this.visited = visited;
        }
    }
}
```

Graph.cs

TestGraph.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    public class TestGraph
    {
        public static void Main(String [] args)
        {
            Graph graph = new Graph (5);
            graph.AddVertex("A" );
            graph.AddVertex("B" );
            graph.AddVertex("C" );
            graph.AddVertex("D" );
            graph.AddVertex("E" );
            graph.AddEdge(0, 1);
            graph.AddEdge(0, 2);
            graph.AddEdge(0, 3);
            graph.AddEdge(1, 2);
            graph.AddEdge(1, 3);
            graph.AddEdge(2, 3);
            graph.AddEdge(3, 4);
            // Two-dimensional array traversal output vertex edge
            and adjacent array
            PrintGraph(graph);
            Debug .Write("\nnBreadth-first search traversal output :
\n" );
            graph.breadthFirstSearch();
        }
        public static void PrintGraph(Graph graph)
        {
            Debug .Write("Two-dimensional array traversal vertex
            edge and adjacent array : \n  ");
            for (int i = 0; i < graph.GetVertexs().Length; i + +)
            {
                Debug .Write(graph.GetVertexs()[i].data + "  ");
            }
        }
    }
}
```

```

        Debug.WriteLine("\n");
        for (int i = 0; i <
graph.GetAdjacencyMatrix().GetLength(0); i++)
        {
            Debug.WriteLine(graph.GetVertexes()[i].data + " ");
            for (int j = 0; j <
graph.GetAdjacencyMatrix().GetLength(0); j++)
            {
                Debug.WriteLine(graph.GetAdjacencyMatrix()[i, j] +
" ");
            }
            Debug.WriteLine("\n");
        }
    }
}

```

Result:

	A	B	C	D	E
A	0	1	1	1	0
B	0	0	1	1	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

Breadth-first search traversal output :

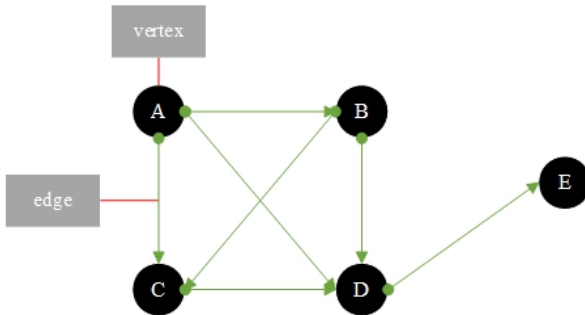
A -> B -> C -> D -> E

Directed Graph Topological Sorting

Directed Graph Topological Sorting:

Sort the vertices in the directed graph with order of direction

Directed Graph has direction : $A \rightarrow B$ and $B \rightarrow A$ are different



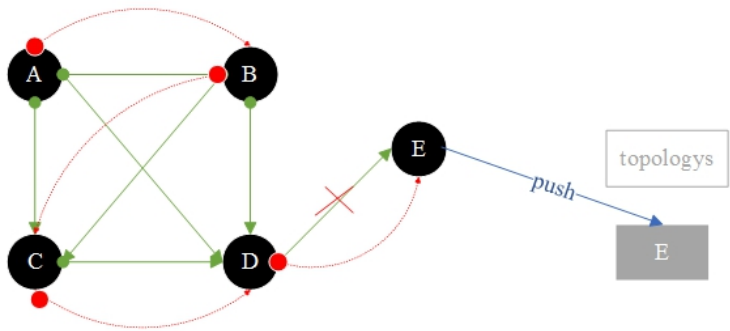
1. The adjacency matrix is described above:

The total number of vertices is a two-dimensional array size, if have value of the edge is **1** , otherwise no value of the edge is **0** .

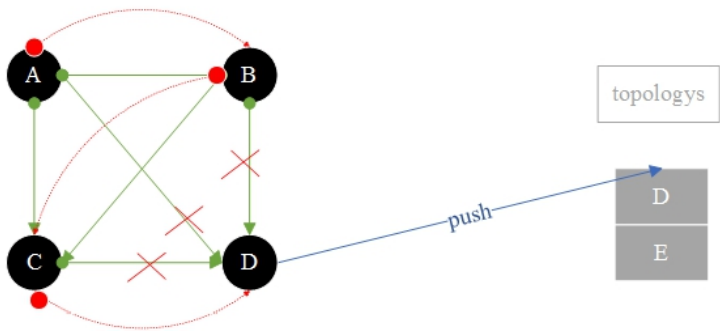
	A	B	C	D	E
A	0	1	1	1	0
B	0	0	1	1	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

Topological sorting from vertex A : **A -> B -> C -> D -> E**

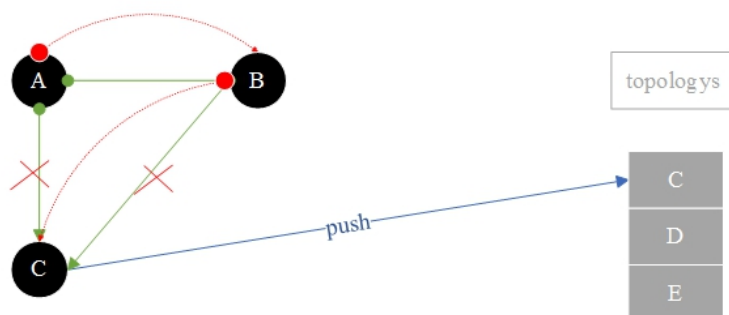
Find no successor vertices E then save to topologys, last E remove from the graph



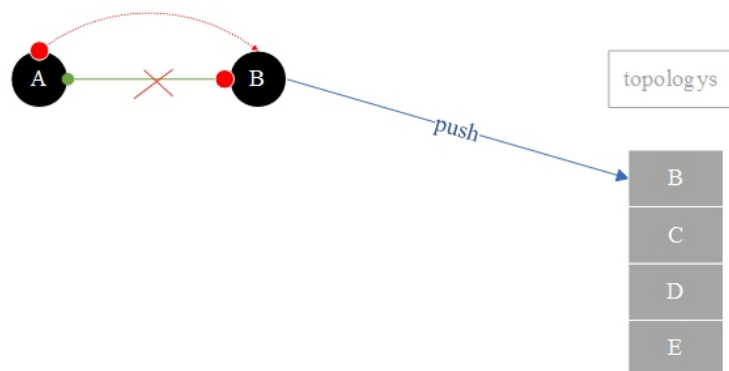
Find no successor vertices D then save to topologys, last D remove from the graph



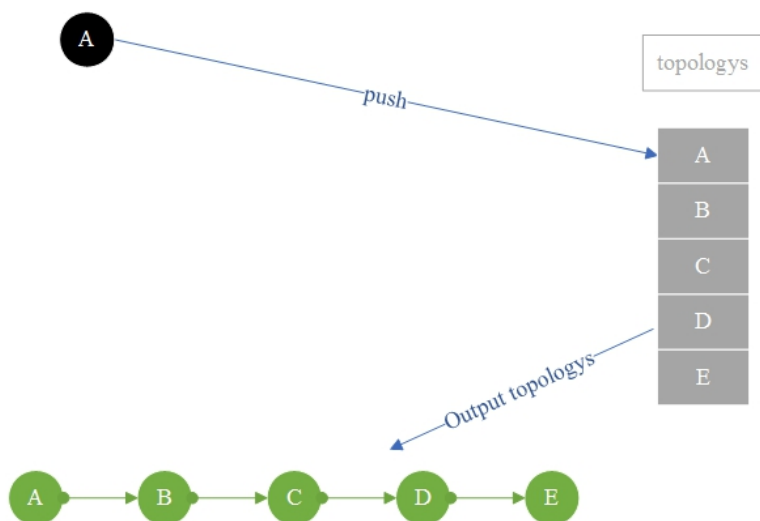
Find no successor vertices C then save to topologys, last C remove from the graph



Find no successor vertices C then save to topologys, last C remove from the graph



Find no successor vertices C then save to topologys, last C remove from the graph



Vertex.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    public class Vertex
    {
        public String data;
        public bool visited; // Have you visited
        public Vertex(String data, bool visited)
        {
            this.data = data;
            this.visited = visited;
        }
    }
}
```

Graph.cs

TestGraph.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
namespace CSharpDatastructuresAlgorithms
{
    public class TestGraph
    {
        public static void Main(String [] args)
        {
            Graph graph = new Graph (5);
            graph.AddVertex("A" );
            graph.AddVertex("B" );
            graph.AddVertex("C" );
            graph.AddVertex("D" );
            graph.AddVertex("E" );
            graph.AddEdge(0, 1);
            graph.AddEdge(0, 2);
            graph.AddEdge(0, 3);
            graph.AddEdge(1, 2);
            graph.AddEdge(1, 3);
            graph.AddEdge(2, 3);
            graph.AddEdge(3, 4);
            // Two-dimensional array traversal output vertex edge
            and adjacent array
            PrintGraph(graph);
            Debug .Write("\nDepth-First Search traversal output : \n"
);
            Debug .WriteLine("Directed Graph Topological Sorting:"
);
            graph.TopologySort();
            for (int i = 0; i < graph.GetTopologys().Length; i++)
            {
                Debug .Write(graph.GetTopologys()[i].data + " -> "
);
            }
        }
        public static void PrintGraph(Graph graph)
```

```

{
    Debug .Write("Two-dimensional array traversal vertex
edge and adjacent array : \n ");
    for (int i = 0; i < graph.GetVertexs().Length; i++)
    {
        Debug .Write(graph.GetVertexs()[i].data + " ");
    }
    Debug .Write("\n");
    for (int i = 0; i <
graph.GetAdjacencyMatrix().GetLength(0); i++)
    {
        Debug .Write(graph.GetVertexs()[i].data + " ");
        for (int j = 0; j <
graph.GetAdjacencyMatrix().GetLength(0); j++)
        {
            Debug .Write(graph.GetAdjacencyMatrix()[i, j] +
" ");
        }
        Debug .Write("\n");
    }
}
}
}
}

```

Result:

Two-dimensional array traversal output vertex edge and adjacent array :

	A	B	C	D	E
A	0	1	1	1	0
B	0	0	1	1	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

Depth-First Search traversal output :

Directed Graph Topological Sorting:

A -> B -> C -> D -> E ->

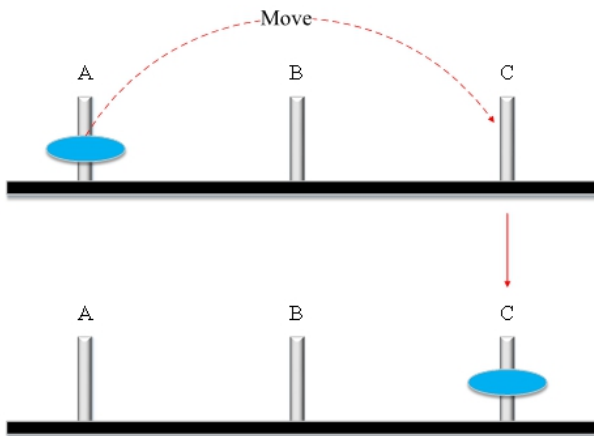
Towers of Hanoi

The Hanoi Tower that a Frenchman M. Claus (Lucas) from Thailand to France in 1883. Hanoi Tower which is supported by three diamond Pillars. At the beginning, God placed 64 gold discs from top to bottom on the first Pillar. God ordered the monks to move all gold discs from the first Pillar to the third Pillar. The principle of large plates under small plates during the handling process. If only one plate is moved daily, the tower will be destroyed. when all the discs are moved that is the end of the world.

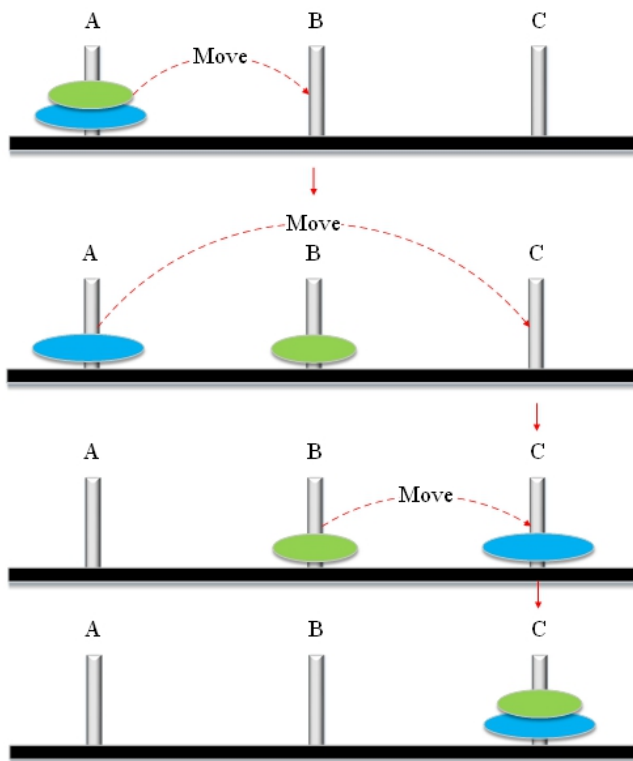
Let's turn this story into an algorithm:

Mark the three columns as ABC.

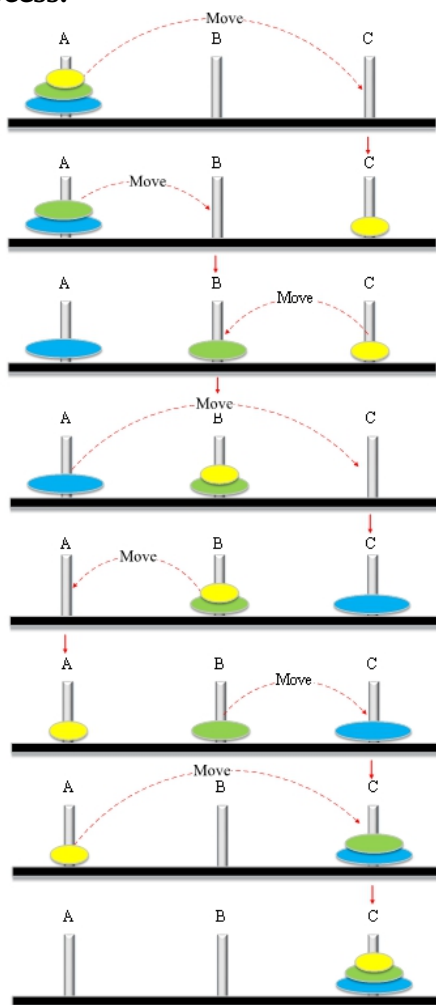
1. If there is only one disc, move it directly to C (**A->C**).
 2. When there are two discs, use B as an auxiliary (**A->B, A->C, B->C**).
 3. If there are more than two discs, use B as an auxiliary(**A->B, A->C, B->C**), and continue to recursive process.
- 1. If there is only one disc, move it directly to C (**A->C**).**



2. When there are two discs, use B as an auxiliary ($A \rightarrow B$, $A \rightarrow C$, $B \rightarrow C$).



3. If more than two discs, use B as an auxiliary, and continue to recursive process.



TowersOfHanoi.cs

```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class TowersOfHanoi
    {
        public static void Main(String [] args)
        {
            Debug.WriteLine("Please enter the number of discs :");
            int n = Convert.ToInt32(Console.ReadLine());
            Hanoi(n, 'A', 'B', 'C');
        }
        public static void Hanoi(int n, char A, char B, char C)
        {
            if (n == 1)
            {
                Debug.WriteLine("Move " + n + " " + A + " to " +
C);
            }
            else
            {
                hanoi(n - 1, A, C, B); // Move the n-1th disc on the
A through C to B
                Debug.WriteLine("Move " + n + " from " + A +
" to " + C);
                hanoi(n - 1, B, A, C); //Move the n-1th disc on the
B through A to C
            }
        }
    }
}
```


Result:

Please enter the number of discs :

1

Move 1 A to C

Please enter the number of discs :

2

Move 1 A to B

Move 2 from A to C

Move 1 B to C

Please enter the number of discs :

3

Move 1 A to C

Move 2 from A to B

Move 1 C to B

Move 3 from A to C

Move 1 B to A

Move 2 from B to C

Move 1 A to C

Fibonacci

Fibonacci : a European mathematician in the 1200s, in his writings: "If there is a rabbit
After a month can birth to a newborn rabbit. At first there was only 1 rabbit, after one month still 1 rabbit. after two month 2 rabbit, and after three months there are 3 rabbit

for example: 1, 1, 2, 3, 5, 8, 13 ...

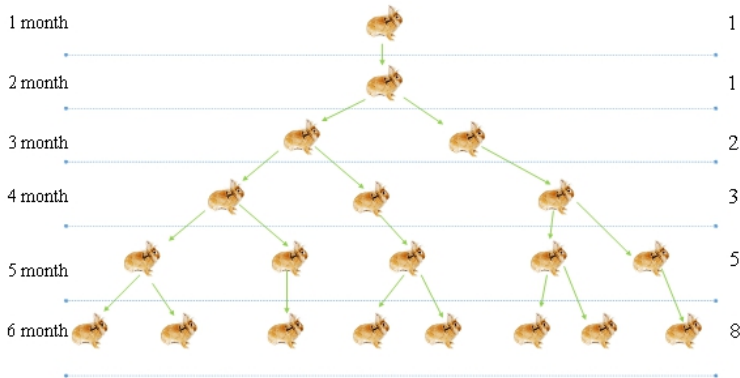
Fibonacci definition:

if $n = 0, 1$

$fn = n$

if $n > 1$

$fn = fn-1 + fn-2$



TestFibonacci.cs

```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class Fibonacci
    {
        public static void Main(String [] args)
        {
            Debug.WriteLine("Please enter the number of month :");
            int number = Convert.ToInt32(Console.ReadLine());
            for (int i = 1; i <= number; i++)
            {
                Debug.WriteLine(i + " month : " + Fibonacci(i));
            }
        }
        public static int Fibonacci(int n)
        {
            if (n == 1 || n == 2)
            {
                return 1;
            }
            else
            {
                return fibonacci(n - 1) + fibonacci(n - 2);
            }
        }
    }
}
```

Result:

Please enter the number of month :

7

1 month : 1

2 month : 1

3 month : 2

4 month : 3

5 month : 5

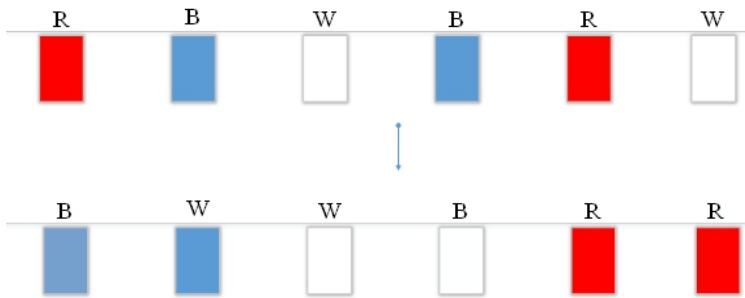
6 month : 8

7 month : 13

Dijkstra

The tricolor flag was originally raised by E.W. Dijkstra, who used the Dutch national flag (Dijkstra is Dutch).

Suppose there is a rope with red, white, and blue flags. At first all the flags on the rope are not in order. You need to arrange them in the order of **blue -> white -> red**. How to move them with the least times. you just only do this on the rope, and only swap two flags at a time.



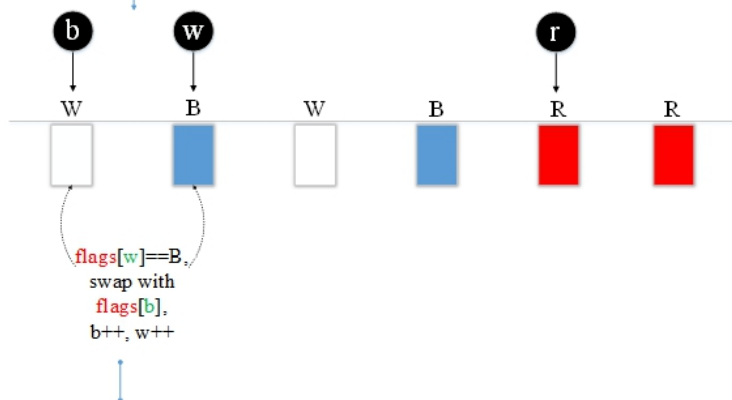
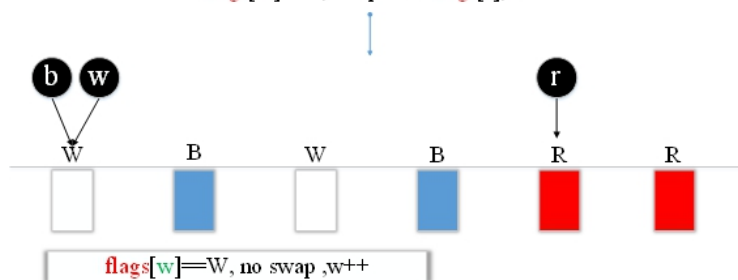
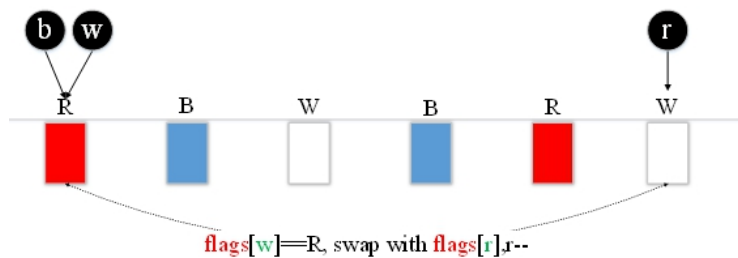
Solution:

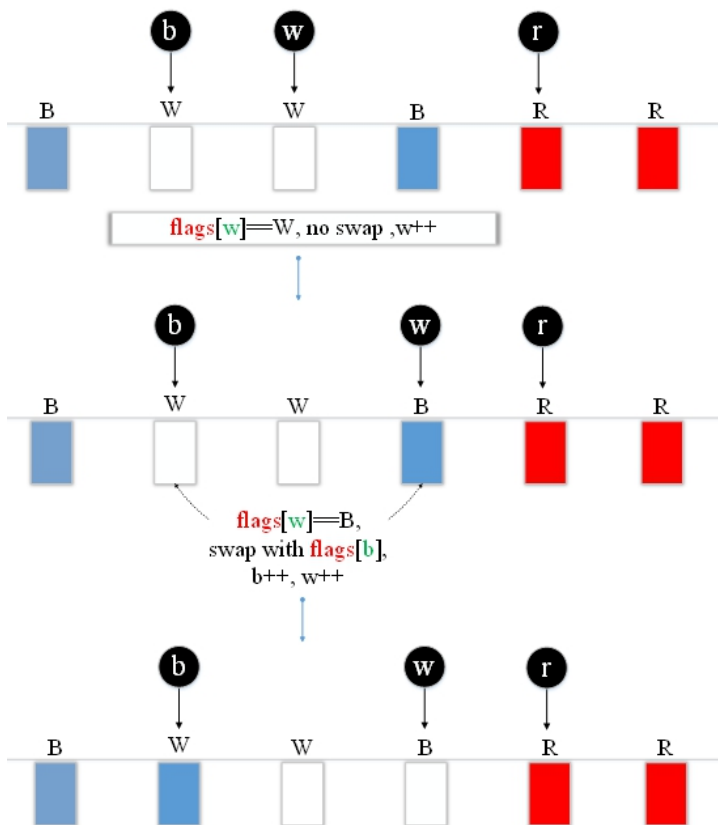
Use the **char arrays** to store the **flags** . For example, **b** , **w** , and **r** indicate the position of **blue** , **white** and **red** flags. The beginning of **b** and **w** is 0 of the array, and **r** is at the end of the array.

- (1) If the position of **w** is a blue flag, **flags [w]** exchange with **flags [b]**. And **whiteIndex** and **b** is moved backward by 1.
- (2) If the position of **w** is a white flag, **w** moves backward by 1.
- (3) If the position of **w** is a red flag, **flags [w]** exchange with **flags [r]**. **r** moves forward by 1.

In the end, the flags in front of **b** are all blue, and the flags behind **r** are all red.

Graphic Solution





Dijkstra.cs

```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class Dijkstra
    {
        public static void Main(String [] args)
        {
            char [] flags = { 'R' , 'B' , 'W' , 'B' , 'R' , 'W' };
            int b = 0, w = 0, r = flags.Length - 1;
            int count = 0;
            while (w <= r) {
                if (flags[w] == 'W' ) {
                    w++;
                } else if (flags[w] == 'B' ) {
                    char temp = flags[w];
                    flags[w] = flags[b];
                    flags[b] = temp;
                    w++;
                    b++;
                    count++;
                } else if (flags[w] == 'R' ) {
                    char m = flags[w];
                    flags[w] = flags[r];
                    flags[r] = m;
                    r--;
                    count++;
                }
            }
            for (int i = 0; i < flags.Length; i++) {
                Debug.Write(flags[i]);
            }
            Debug.WriteLine("\nThe total exchange count : " + count);
        }
    }
}
```

Result:

BBWWRR

The total exchange count : 4

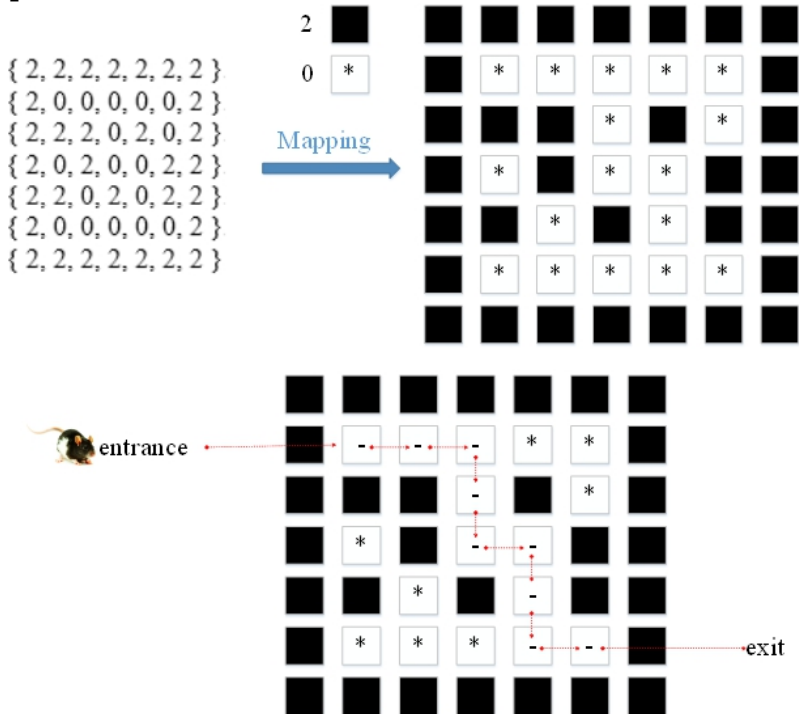
Mouse Walking Maze

Mouse Walking Maze is a basic type of recursive solution. We use **2** to represent the **wall** in a **two-dimensional array** , and use **1** to represent the **path** of the mouse, and try to find the path from the entrance to the exit.

Solution:

The mouse moves in four directions: **up, left, down, and right** . if hit the **wall** go back and select the next forward direction, so test the four directions in the array until mouse reach the exit.

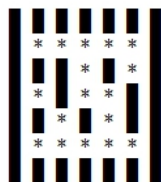
Graphic Solution



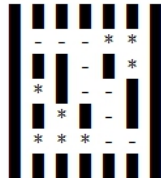
MouseWalkingMaze.cs

Result:

Maze:



Maze Path :



Eight Coins

There are eight coins with the same appearance, one is a counterfeit coin, and the weight of counterfeit coin is different from the real coin, but it is unknown whether the counterfeit coin is lighter or heavier than the real coin. Please design an efficient algorithm to detect this counterfeit coin.

Solution:

Take six **a, b, c, d, e, f** from eight coins, and put three to the balance for comparison. Suppose **a, b, c** are placed on one side, and **d, e, f** are placed on the other side.

1. $a + b + c > d + e + f$
2. $a + b + c = d + e + f$
3. $a + b + c < d + e + f$

If $a + b + c > d + e + f$, there is a counterfeit coin in one of the six coins, and **g, h** are real coins. At this time, one coin can be removed from both side. Suppose that **c and f** are removed. At the same time, one coin at each side is replaced. Suppose the coins **b and e** are interchanged, and then the second comparison. There are also three possibilities:

1. $a + e > d + b$: the counterfeit currency must be one of **a, d**. as long as we compare a real currency **h** with **a**, we can find the counterfeit currency. If $a > h$, **a** is a heavier counterfeit currency; if $a = h$, **d** is a lighter counterfeit currency.

2. $a + e = d + b$: the counterfeit currency must be one of **c, f**, and the real coin **h** is compared with **c**. If $c > h$, **c** is a heavier counterfeit currency; if $c = h$, then **f** is a lighter counterfeit currency.

3. $a + e < d + b$: one of **b or e** is a counterfeit coin, Also use the real coin **h** to compare with **b**, if $b > h$, then **b** is a heavier counterfeit currency; if $b = h$, then **e** is a lighter counterfeit currency;

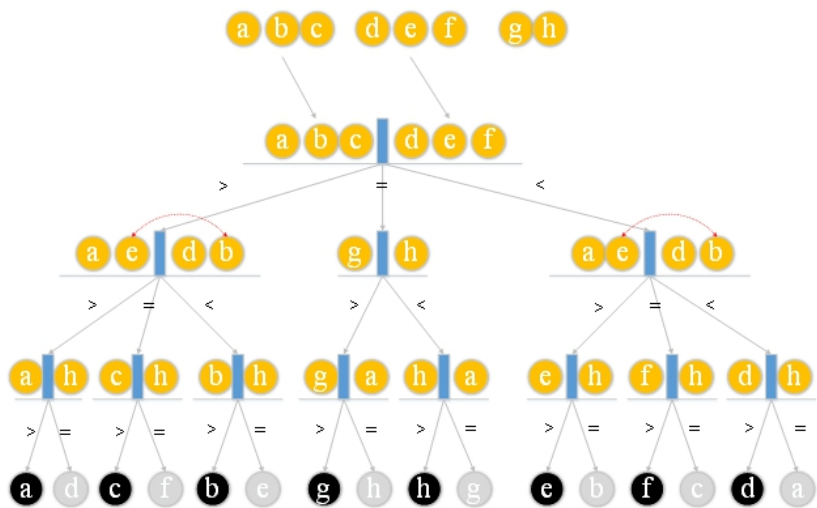
Graphic Solution



Heavy



Light



TestEightCoins.cs

```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class TestEightcoins
    {
        public static void Compare(int [] coins, int i, int j, int k) { //
coin[k] true, coin[i] > coin[j]
            if (coins[i] > coins[k]) //
coin[i] > coin[j] && coin[i] > coin[k] -----> coin[i] is a heavy
counterfeit coin
                Debug.WriteLine("\nCounterfeit currency " + (i + 1)
+ " is heavier ");
            else // coin[j] is a light counterfeit coin
                Debug.WriteLine("\nCounterfeit currency " + (j + 1)
+ " is lighter ");
        }
        public static void Eightcoins(int [] coins)
        {
            if (coins[0] + coins[1] + coins[2] == coins[3] +
coins[4] + coins[5]) { //(a + b + c) == (d + e + f)
                if (coins[6] > coins[7]) //g > h? (g > a?g:a):(h > a?h:a)
                    Compare(coins, 6, 7, 0);
                else //h > g? (h > a?h:a):(g > a?g:a)
                    Compare(coins, 7, 6, 0);
            } else if (coins[0] + coins[1] + coins[2] > coins[3] +
coins[4] + coins[5]) { //(a + b + c) > (d + e + f)
                if (coins[0] + coins[3] == coins[1] + coins[4]) //
(a + e) == (d + b)
                    Compare(coins, 2, 5, 0);
                else if (coins[0] + coins[3] > coins[1] + coins[4])
//(a + e) > (d + b)
                    Compare(coins, 0, 4, 1);
                if (coins[0] + coins[3] < coins[1] + coins[4]) //(a
+ e) < (d + b)
                    Compare(coins, 1, 3, 0);
            } else if (coins[0] + coins[1] + coins[2] < coins[3] +
coins[4] + coins[5]) { //(a + b + c) < (d + e + f)
```

```

        if (coins[0] + coins[3] == coins[1] + coins[4]) //
(a + e) > (d + b)
            Compare(coins, 5, 2, 0);
        else if (coins[0] + coins[3] > coins[1] + coins[4])
// (a + e) > (d + b)
            Compare(coins, 3, 1, 0);
        if (coins[0] + coins[3] < coins[1] + coins[4]) // (a
+ e) < (d + b)
            Compare(coins, 4, 0, 1);
    }
}
public static void Main(String [] args)
{
    int [] coins = new int [8];
    // Initial coin weight is 10
    for (int i = 0; i < 8; i++)
        coins[i] = 10;
    Debug.WriteLine("Enter weight of counterfeit
currency (larger or smaller than 10) :");
    Random rn = new Random ();
    coins[rn.Next(8)] = Convert.ToInt32(Console
.ReadLine());
    Eightcoins(coins);
    for (int i = 0; i < 8; i++)
        Debug.Write(coins[i] + " , ");
    }
}
}

```

Result:

First run:

Enter weight of counterfeit currency (larger or smaller than 10) :
2

Counterfeit currency 2 is lighter
10 , 2 , 10 , 10 , 10 , 10 , 10 , 10 ,

Run again:

Enter weight of counterfeit currency (larger or smaller than 10) :
13

Counterfeit currency 4 is heavier
10 , 10 , 10 , 13 , 10 , 10 , 10 , 10 ,

Knapsack Problem

Suppose you have a backpack with a weight of up to 8 kg, and you want to the backpack with a total price of Products, suppose the fruit (ID, Name, Price and Weight)

			Weight		
0500					
1000					
0250					
3000					
0700					

Solution:

To solve the optimization problem we can use **Dynamic Programming** . In the beginning there is a empty set, every time add an element, find the best solution at this stage, until all elements are added. After entering the set finally can get the best solution.

There are two arrays, **value and item**

value: the total price of the current best solution.

item : the last fruit in the backpack.

there are 8 backpacks with a weight of 1 to 8, and find the best solution for each backpack.

Gradually put the fruit in the backpack and find the best solution:

1. Put in plums:

					Weight	1kg	Backpack					
						000						
						000						
						2000	Plum					

2. Put in apples:

					Weight	1kg	Backpack					
						000						
						000						
						2000	Apple					

3. Put in oranges:

					Weight	Backpack						
						8123						
						9050						
						090						
						2118						
						Apple						

4. Put in strawberries:

					Weight	Backpack						
						8123						
						9050						
						10023						
						Plus Strawberry						
						Strawberry						
						Apple						

5. Put in melons:

					Weight	Backpack						
						8123						
						9050						
						10023						
						Oranges Strawberry						
						Strawberry						

From the last table that when the backpack weighs 8 kg, a maximum of 9050, so the best solution is to put **Strawberries, Oranges and Apples** , and the total price is 9050.

TestKnapsack.cs

```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class Fruit
    {
        private String name;
        private int size;
        private int price;
        public Fruit(String name, int size, int price) {
            this.name = name;
            this.size = size;
            this.price = price;
        }
        public String getName() {
            return name;
        }
        public int getPrice() {
            return price;
        }
        public int getSize() {
            return size;
        }
    }
    class TestKnapsack
    {
        public const int MAXSIZE = 8;
        public const int MINSIZE = 1;
        public static void Main(String [] args) {
            int [] item = new int [MAXSIZE + 1];
            int [] value = new int [MAXSIZE + 1];
            Fruit [] fruits = {
                new Fruit ("Plum ", 4, 4500),
                new Fruit ("Apple", 5, 5700),
                new Fruit ("Orange", 2, 2250),
                new Fruit ("Strawberry", 1, 1100),
                new Fruit ("Melon" , 6, 6700) };
            for (int i = 0; i < fruits.Length; i++) {
```

```

        for (int j = fruits[i].getSize(); j <= MAXSIZE; j++)
        {
            int p = j - fruits[i].getSize();
            int newValue = value[p] + fruits[i].getPrice();
            if (newValue > value[j]) { // Find the best
solution
                value[j] = newValue;
                item[j] = i;
            }
        }
    }
    Debug.WriteLine("Item \t Price" );
    for (int i = MAXSIZE; i >= MINSIZE; i = i -
fruits[item[i]].getSize()) {
        Debug.WriteLine(fruits[item[i]].getName() + "\t" +
fruits[item[i]].getPrice());
    }
    Debug.WriteLine("Total \t" + value[MAXSIZE]);
}
}
}
}

```

Result:

Item	Price
Strawberry	1100
Orange	2250
Apple	5700

Josephus Problem

There are 9 Jewish hid in a hole with Josephus and his friends . The 9 Jews decided to die rather than be caught by the enemy, so they decided In a suicide method, 11 people are arranged in a circle, and the first person reports the number. After each number is reported to the third person, the person must commit suicide. Then count again from the next one until everyone commits suicide. But Josephus and his friends did not want to obey. Josephus asked his friends to pretend to obey, and he arranged the friends with himself. In the 2th and 7st positions, they escaped this death game.

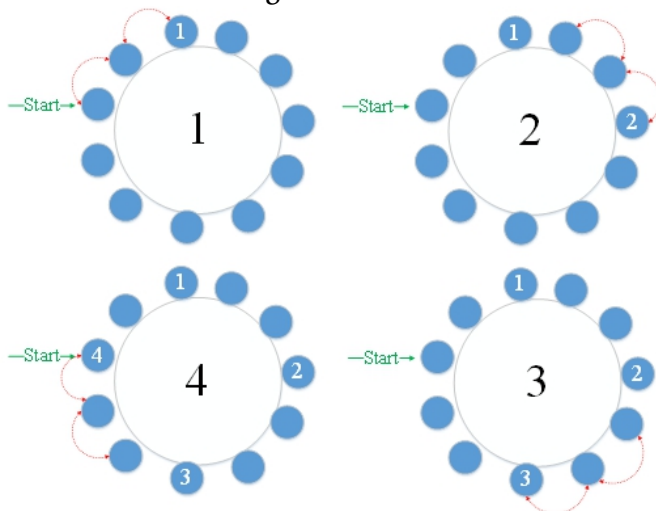
Solution:

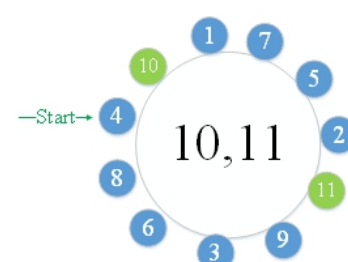
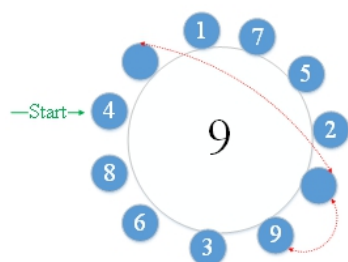
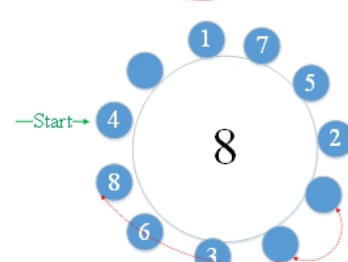
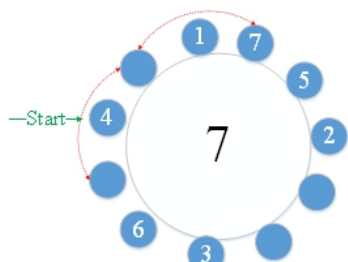
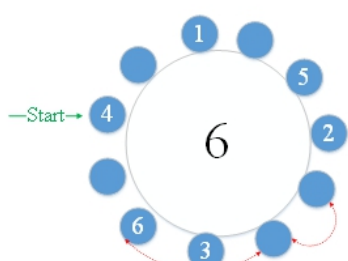
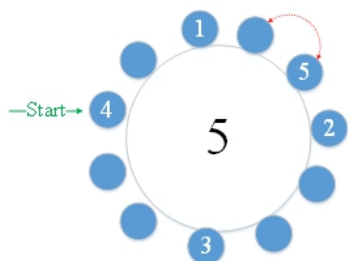
As long as the array is treated as a ring. Fill in a count for each dataless area, until the count reaches 11, and then list the array from index 1, you can know that each suicide order in this position is the Joseph's position. The 11-person position is as follows:

4 10 1 7 5 2 11 9 3 6 8

From the above, the last two suicide was in the 31st and 16th position. The previous one

Everyone died, so they didn't know that Joseph and his friends didn't follow the rules of the game.





Joseph.cs

```
using System;
using System.Diagnostics;
namespace CSharpDatastructuresAlgorithms
{
    class Joseph
    {
        private const int N = 11;
        private const int M = 3;
        public static void Main(String [] args) {
            int [] man = new int [N];
            int count = 1;
            int i = 0, pos = -1;
            int alive = 0;
            while (count <= N) {
                do {
                    pos = (pos + 1) % N; // Ring
                    if (man[pos] == 0)
                        i++;
                    if (i == M) {
                        i = 0;
                        break ;
                    }
                } while (true );
                man[pos] = count;
                count++;
            }
            Debug.WriteLine("\nJoseph sequence : ");
            for (i = 0; i < N; i++)
                Debug.Write(man[i] + " , ");
        }
    }
}
```

Result:

Joseph sequence :

4 , 10 , 1 , 7 , 5 , 2 , 11 , 9 , 3 , 6 , 8 ,

If you enjoyed this book and found some benefit in reading this, I'd like to hear from you and hope that you could take some time to post a review on Amazon. Your feedback and support will help us to greatly improve in future and make this book even better.

You can follow this link now.

<http://www.amazon.com/review/create-review?&asin=B08F39C488>

=

Different country reviews only need to modify the amazon domain name in the link:

www.amazon.co.uk

www.amazon.de

www.amazon.fr

www.amazon.es

www.amazon.it

www.amazon.ca

www.amazon.nl

www.amazon.in

www.amazon.co.jp

www.amazon.com.br

www.amazon.com.mx

www.amazon.com.au

I wish you all the best in your future success!